프로그람개발문고

Qt프로그람개발법

새세대

교육위원회 교육정보쎈터

주체99(2010)

차 례

머리말

위대한 령도자 **김정일**동지께서는 다음과 같이 지적하시였다.

《프로그람을 개발하는데서 기본은 우리 식의 프로그람을 개발하는것입니다. 우리는 우리 식의 프로그람을 개발하는 방향으로 나가야 합니다.》 (《**김정일**선집》제15권, 196페지)

위대한 령도자 **김정일**동지의 현명한 령도에 의하여 오늘 우리 나라에서는 프로그람기술이 빠른 속도로 발전하고있다.

우리의 과학자, 연구사들은 우리 식 조작체계 《붉은별》을 개발하였으며 각종 도구들과 응용프로그람들을 개발하기 위한 연구사업을 활발히 진행하고있다.

우리는 정보공학을 전공하는 교원, 연구사들과 대학생들이 프로그람개발도구인 Qt에 의하여 프로그람을 능숙하게 작성할수 있도록 하기 위하여 《Qt일반지식》과 《Qt프로그람개발법》, 《Qt프로그람개발도구》, 《Qt실례프로그람》을 출판한다.

《Qt프로그람개발법》에서는 Qt에 의한 GUI프로그람작성방법을 서술한다.

《Qt일반지식》에서는 Qt의 객체모형, 신호와 처리부, 사건과 사건려과기를 비롯한 Qt 프로그람이 기초하고있는 일반적인 원리들을 서술한다.

《Qt프로그람개발도구》에서는 Qt Designer를 비롯한 Qt에 부속되여있는 각종 도구들의 사용법을 서술한다.

《Qt실례프로그람》에서는 Qt로 작성한 실례프로그람들을 서술한다.

Qt도구묶음(toolkit)은 C++클라스서고로서 여러가동환경GUI프로그람을 건설하기 위한 한조의 도구이다. Qt는 프로그람작성자들이 하나의 원천나무를 리용하여 응용프로그람들을 Windows 95～XP, Mac OS X, Linux, Solaris, HP-UX, 그밖에 X11를 가지는 Unix의 다른 판들에서 실행하게 한다. 한가지 Qt판본은 또한 같은 API를 가지고 Embedded Linux에서도 사용할수 있다.

이 책에서는 Qt 3에 의한 GUI프로그람작성방법에 대하여 설명한다. 이 책에서는 사용자정의창문부품의 작성과 끌어다놓기의 제공과 같은 고급한 수법들을 설명한다.

독자들이 C++에 대한 기초지식을 리해하고있는것을 전제로 한다. 코드실례들은 C++의 부분모임을 사용하지만 Qt프로그람을 작성할 때 드문히 요구되는 C++의 기능들은 설명하지 않는다.

이 책은 18개의 장으로 되였는데 간단한 대화칸만들기로부터 시작하여 입력과 출력, 자료기지작성, 망프로그람작성, XML, 다중스레드실현방법에 대하여 구체적으로 서술한다.

제1장. 시작하기

이 장에서는 표준 C++와 Qt가 제공하는 기능을 결합하여 여러개의 자그마한 도형방식사용자대면부(GUI) 응용프로그람들을 만드는 방법을 보여준다. 또한 Qt의 2가지 기본개념인《신호와 처리부》 그리고 배치에 대하여 소개한다.

제1절. Hello Qt프로그람

여기에 아주 간단한 Qt프로그람이 있다.

```
1 #include <qapplication.h>
2 #include <qlabel.h>
3 int main(int argc, char *argv[])
4 {
5   QApplication app (argc, argv);
6   QLabel *label = new QLabel("Hello Qt!", 0);
7   app.setMainWidget(label);
8   label->show();
9   return app.exec();
10 }
```

우선 프로그람을 한행씩 설명한 다음 프로그람을 콤파일하고 실행하는 방법을 설명한다. 1행과 2행은 QApplication과 QLabel클라스의 정의를 포함한다.

5행은 응용프로그람의 자원을 관리하는 QApplication객체를 생성한다. Qt가 자기의 지령행인수를 가지므로 QApplication구성자는 argc와 argv를 요구한다.

6행은《Hello Qt!》라고 현시하는 QLabel창문부품을 생성한다. Qt에서 창문부품(widget)은 사용자대면부의 시각적요소이다. 단추, 차림표, 흘림띠, 틀은 모두 창문부품의 실례들이다. 창문부품은 다른 창문부품들을 포함할수 있다. 례를 들면 응용프로그람창문은 보통 하나의 QMenuBar, 하나의 QToolBar, 하나의 QStatusBar, 그밖에 다른 창문부품들을 포함하는 창문부품이다. 0인수(null지적자)의 QLabel구성자는 창문부품이 독립적인 창문이고 다른 창문안에 놓이는 창문부품이 아니라는것을 의미한다.

7행은 표식자를 응용프로그람의 기본창문부품으로 만든다. 사용자가 기본창문부품을 닫으면(실례로 창문제목띠의 X를 찰칵하여) 프로그람은 완료된다. 사용자가 창문을 닫아서 기본창문부품이 없는 경우에도 프로그람은 배경에서 계속 실행될수 있다.

8행은 표식자를 표시한다. 창문부품은 늘 은폐된 상태로 생성되므로 그것을 표시하기전에 전용화하여 깜빡거림을 피할수 있다.

9행은 응용프로그람의 조종을 Qt에 넘긴다. 이 시점에서 프로그람은 일종의 대기(stand-by)방식에 들어가며 마우스찰칵이나 건누르기와 같은 사용자의 작용(action)을 기다린다.

사용자의 작용은 사건(통보문이라고도 한다.)을 발생시키며 보통 프로그람은 하나이상의 함수들을 실행하여 사건에 응답할수 있다. 이 시점에서 GUI응용프로그람은 관례적인 묶음(batch)프로그람과 전혀 다르다. 일반적으로 묶음프로그람은 입력을 처리하고 결과를 생성하며 사람들과 교제없이 완료한다.

그림 1-1. Windows XP에서 Hello프로그람

그러면 콤퓨터에서 프로그람을 시험해보자. 우선 Qt를 설치해야 한다. 지금까지는 Qt가 정확히 설치되여있고 Qt의 bin등록부가 PATH환경변수안에 있는것으로 가정하였다. (Windows에서는 이것이 Qt설치프로그람에 의하여 자동적으로 수행되므로 걱정할 필요는 없다.)

또한 hello라는 등록부안의 hello.cpp파일에 Hello프로그람의 원천코드가 있어야 한다. hello.cpp를 자체로 입력한다.

지령재촉문으로부터 등록부를 hello로 변경한 다음 qmake -project라고 입력하여 가동환경(platform)에 의존하지 않는 프로젝트파일(hello.pro)을 만들고 qmake hello.pro라고 입력하여 프로젝트파일로부터 가동환경에 고유한 makefile을 만든다. make를 실행하여 프로그람을 건설하고 Windows에서는 hello, Unix에서는 ./hello, Mac OS X에서는 open hello.app라고 입력하여 프로그람을 실행한다. Visual C++를 리용한다면 make대신에 nmake를 실행하여야 한다. 혹은 qmake -tp vc hello.pro라고 입력하여 hello.pro로부터 Visual Studio프로젝트파일을 생성하고 프로그람을 Visual Studio에서 건설할수 있다.

그림 1-2. 기본HTML형식의 표식자

간단한 HTML형식을 사용하여 표식자를 밝게 할수 있다. 다음 행

```
QLabel *label = new QLabel("Hello Qt!", 0);
```

을

```
QLabel *label = new QLabel("<h2><i>Hello</i> " "<font color=red>Qt!</font></h2>", 0);
```

로 변경하고 응용프로그람을 다시 건설한다.

제2절. 련결의 만들기

다음의 실례는 사용자의 작용에 응답하는 방법을 보여준다. 응용프로그람은 사용자가 마우스를 찰칵하면 중지되는 하나의 단추로 이루어진다. 원천코드는 Hello와 거의 비슷하지만 기본창문부품으로서 QLabel대신에 QPushButton을 사용하며 사용자의 작용(단추를 찰칵하여)과 코드의 한 부분을 련결한다.

그림 1-3. Quit응용프로그람

```
1 #include <qapplication.h>
2 #include <qpushbutton.h>
3 int main(int argc, char *argv[])
4 {
5    QApplication app(argc, argv);
6    QPushButton *button = new QPushButton("Quit", 0);
7    QObject::connect(button, SIGNAL(clicked()),
8          &app, SLOT(quit()));
9    app.setMainWidget(button);
10   button->show();
11   return app.exec();
12 }
```

Qt의 창문부품은 신호(signal)를 발생(emit)하여 사용자의 작용이 있거나 상태의 변화가 발생하였다는것을 알린다. (Qt신호는 Unix신호와 관계없다. 이 책에서는 Qt신호에 대해서만 취급한다.) 실례로 QPushButton은 사용자가 단추를 찰칵할 때 clicked()신호를 발생한다. 신호를 처리부(slot)라고 부르는 하나의 함수와 련결하면 신호가 발생될 때 처리부가 자동적으로 실행된다. 실례에서는 단추의 clicked()신호를 QApplication객체의 quit()처리부에 련결한다. SIGNAL()과 SLOT()마크로는 문법의 한 부분으로서 다음 장에서 자세히 설명한다.

그러면 응용프로그람을 만들어보자. quit.cpp가 들어있는 quit라는 등록부를 만들었다고 가정하자. quit등록부에서 qmake를 실행하여 프로젝트파일을 생성한 다음 qmake를 다시 실행하여 makefile을 생성한다.

```
qmake -project
qmake quit.pro
```

이제는 응용프로그람을 건설하고 실행한다. Quit를 찰칵하거나 Space건을 눌러서 프로그람을 완료한다.

다음의 실례에서는 신호와 처리부를 사용하여 두개의 창문부품을 동기시키는 방법을 보여준다. 응용프로그람이 사용자의 나이를 물으면 사용자는 스핀칸이나 미끄럼칸(slider)을 조작하여 입력할수 있다.

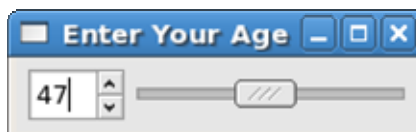



그림 1-4. 나이입력응용프로그람

응용프로그람은 3개의 창문부품 즉 하나의 QSpinBox, 하나의 QSlider 그리고 하나의 QHBox(수평배치칸)로 이루어진다. QHBox는 응용프로그람의 기본창문부품이다. QSpinBox와 QSlider는 QHBox안에 그려지며 QHBox의 자식들이다.

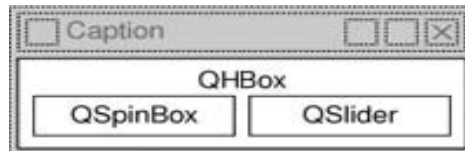



그림 1-5. 나이입력응용프로그람의 창문부품들

```
1 #include <qapplication.h>
2 #include <qhbox.h>
3 #include <qslider.h>
4 #include <qspinbox.h>
5 int main(int argc, char *argv[])
6 {
7    QApplication app(argc, argv);
8    QHBox *hbox = new QHBox(0);
9    hbox->setCaption("Enter Your Age");
10   hbox->setMargin(6);
11   hbox->setSpacing(6);
12   QSpinBox *spinBox = new QSpinBox(hbox);
13   QSlider *slider = new QSlider(Qt::Horizontal, hbox);
14   spinBox->setRange(0, 130);
15   slider->setRange(0, 130);
16   QObject::connect(spinBox, SIGNAL(valueChanged(int)),
17          slider, SLOT(setValue(int)));
18   QObject::connect(slider, SIGNAL(valueChanged(int)),
19          spinBox, SLOT(setValue(int)));
20   spinBox->setValue(35);
21   app.setMainWidget(hbox);
22   hbox->show();
23   return app.exec();
24 }
```

8～11행은 QHBox를 설정한다.(QHBox구성자에서 콤파일러오유가 발생하면 이것은 낡은 판의 Qt를 사용하고있다는것을 의미한다. 이때 자기가 Qt 3.2.0이나 그 이후의 Qt 3출하판을

사용하고있는가 확인한다.) 창문의 제목띠에 현시할 본문을 설정하기 위하여 setCaption()을 호출한다. 그때 자식창문부품들사이에 약간한 공간(6화소)을 둔다.

12행과 13행은 QHBox를 부모로 하여 QSpinBox와 QSlider를 각각 하나씩 생성한다.

창문부품들의 위치나 크기를 명백히 설정하였어도 QSpinBox와 QSlider는 QHBox안에 보기좋게 나란히 배치된다. 이것은 QHBox가 자식들의 요구에 기초하여 자동적으로 합리적인 위치와 크기를 그것들에 할당하기때문이다. Qt는 QHBox와 같은 클라스들을 수많이 제공하여 프로그람작성자들이 화면위치와 관련한 어려운 코드를 작성하는 자질구레한 일에서 해방시켜준다.

14행과 15행은 스핀칸과 미끄럼칸에 유효범위를 설정한다. (여기서는 130살미만이라고 가정한다.) 16～19행에 보여주는 2개의 connect()호출은 스핀칸과 미끄럼칸이 늘 같은 값을 표시하도록 동기된다는것을 담보한다. 한 창문부품의 값이 변화될 때마다 valueChanged(int)신호가 발생되고 다른 창문부품의 setValue(int)처리부가 새 값으로 호출된다.

20행은 스핀칸의 값을 35로 설정한다. 이때 QSpinBox는 int값 35를 인수로 하여 valueChanged(int)신호를 발생시킨다. 이 인수는 QSlider의 setValue(int)처리부에 넘어가며 미끄럼칸의 값을 35로 설정한다. 그때 미끄럼칸은 valueChanged(int)신호를 발생시키며 그 자체의 값이 변경되였으므로 스핀칸의 setValue(int)처리부를 실행한다. 그러나 이 시점에서 setValue(int)는 스핀칸의 값이 이미 35이므로 신호를 발생시키지 않는다. 이것은 무한재귀를 방지한다. 그림 1-6은 이러한 상황을 요약한다.

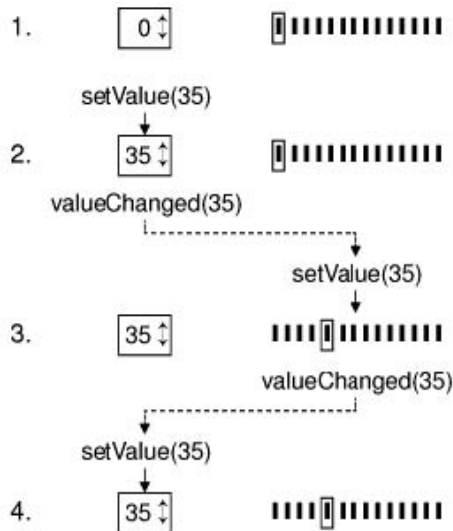



그림 1-6. 한개 값의 변경에 의한 두개 값의 변경

22행은 QHBox와 그의 두 자식창문부품을 표시한다.

Qt에서 사용자대면부를 건설하는 수법은 리해하기 쉽고 유연성이 아주 높다. Qt프로그람작성자들이 사용하는 가장 일반적인 수법은 필요한 창문부품들의 실례를 만든 다음 요구될 때 그 속성들을 설정하는것이다. 프로그람작성자들은 창문부품들에 배치관리자를 추가하여

크기와 위치를 자동적으로 조절한다. Qt의 신호-처리부기구에 의하여 창문부품들을 서로 련결하여 사용자대면부의 동작을 관리한다.

제3절. 참고문서의 사용

Qt의 참고문서는 Qt개발자들에게 아주 중요한 문서이다. 여기서는 Qt의 클라스와 함수들을 모두 설명한다. (Qt 3.2에는 400개이상의 공개클라스와 6000개이상의 함수들이 포함되여있다.) 이 책에서는 Qt의 수많은 클라스와 함수들의 사용방법을 기본적으로 설명하지만 그것들을 모두 언급하지 않는다.

창문부품의 형식

지금까지 보아온 그림들은 Windows XP에서 취한것이지만 Qt응용프로그람은 유지하고있는 모든 가동환경에서 아주 자연스럽게 표시할수 있다. Qt는 특정한 가동환경이나 도구묶음의 창문부품일식을 포함하는것이 아니라 가동환경의 형식(style)을 모의함으로써 이것을 달성한다.

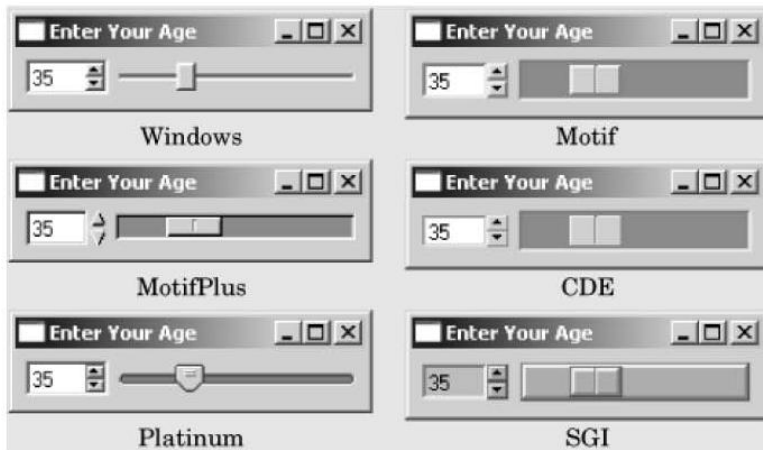



그림 1-7. 어디서나 사용할수 있는 형식들

Qt응용프로그람의 사용자들은 –style지령행선택을 사용하여 기정형식을 무시할수 있다. 례를 들면 Unix에서 나이입력응용프로그람을 Platinum형식으로 기동하려면 지령행에서 간단히 다음과 같이 입력한다.

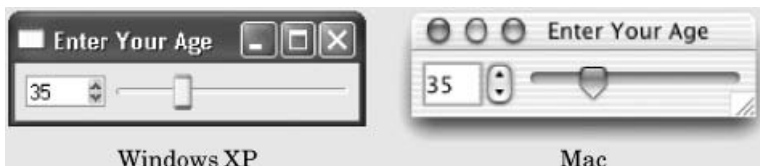
```
./age –style=Platinum
```


그림 1-8. 가동환경에 고유한 형식들

다른 형식들과 달리 Windows XP와 Mac의 형식들은 가동환경의 주제(theme)엔진에 기초하고있으므로 자기 본래의 가동환경에서만 사용할수 있다.

참고문서는 Qt의 doc\html등록부에 HTML형식으로 존재하며 웨브열람기에 의하여 읽을수

있다. 또한 Qt의 방조열람기인 Qt Assistant를 사용할수 있다. Qt Assistant는 웨브열람기보다 더 빠르고 편리하며 강력한 탐색기능과 색인기능을 갖추고있다. Qt Assistant를 호출하려면 Windows에서는 Start차림표에서 Qt 3.3.x|Qt Assistant를 찰칵하고 Unix에서는 지령행에 assistant라고 입력하며 Mac OS X Finder에서 assistant를 두번 찰칵한다.

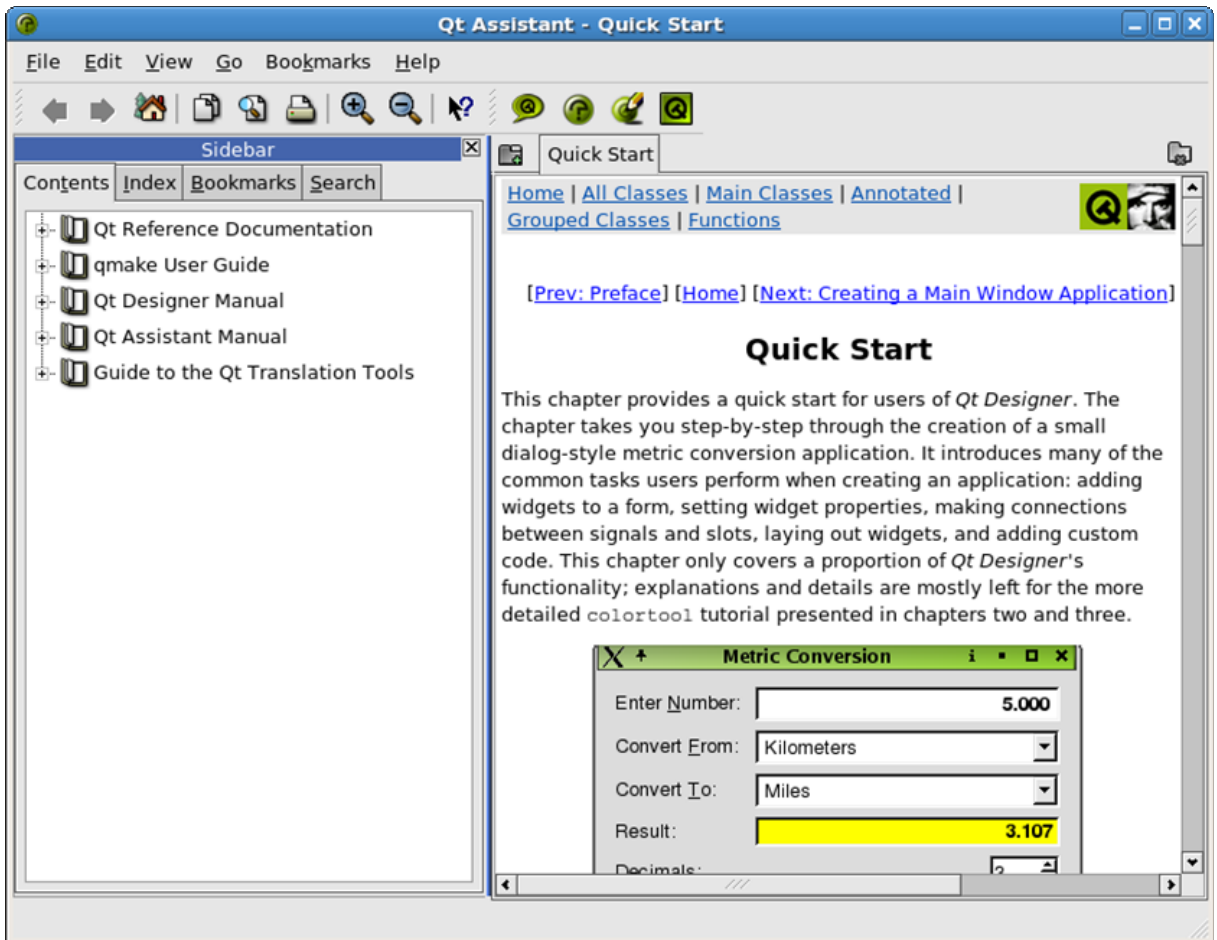



그림 1-9. Qt Assistant에 펼쳐진 Qt문서

홈페지의 《API Reference》 의 항목들은 Qt의 클라스들을 항행하는 여러가지 방법을 제공한다. 《All Classes》페지에는 Qt API의 모든 클라스들을 보여준다. 《Main Classes》페지에는 가장 일반적으로 사용하는 Qt클라스들만 보여준다. 련습삼아 이 장에서 사용한 클라스와 함수들을 찾아볼수 있다. 계승되는 함수들은 기초클라스의 문서에서 설명한다. 례를 들면 QPushButton에는 자체의 show()함수가 없지만 그 선조인 QWidget로부터 계승된다. 그림 1-10에는 지금까지 고찰해온 클라스들사이의 관계를 보여준다.

Qt
QObject
QApplication QWidget
QButton QFrame QSlider QSpinBox
QPushButton QHBox QLabel

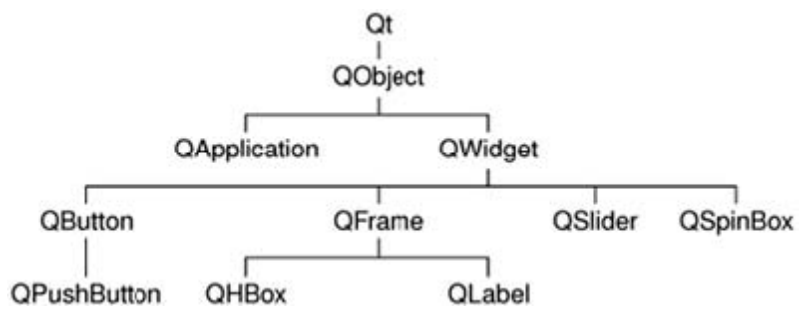



그림 1-10. 지금까지 고찰한 Qt클라스들의 계승나무

제2장. 대화칸의 만들기

이 장에서는 Qt에 의하여 대화칸(dialog box)을 만드는 방법을 설명한다. 대화칸은 사용자와 응용프로그람사이에 서로 교제하는 수단이다.

대화칸은 사용자에게 자기가 좋아하는 값들을 선택할 기회를 준다. 대부분의 GUI응용프로그람들은 차림표띠와 도구띠를 가지는 하나의 기본창문과 기본창문을 보충하는 한 조의 대화칸으로 이루어진다. 또한 적당한 작용들을 수행함으로써 사용자의 선택에 직접 응답하는 대화칸프로그람들을 작성할수 있게 한다(례를 들면 수산기프로그람).

우선 순수 코드를 쓰는 방법으로 첫 대화칸을 만들고 그 동작을 고찰하기로 한다. 그다음 Qt의 시각적인 설계도구인 Qt Designer에 의하여 대화칸을 건설하는 방법을 고찰한다. Qt Designer를 사용하면 코드를 작성하는것보다 훨씬 더 빠르고 각이한 설계들을 간단히 시험하고 변경할수 있다.

제1절. QDialog의 파생클라스 만들기

처음의 실례는 완전히 C++로 쓴 Find대화칸이다. 대화칸을 클라스로서 실현함으로써 대화칸을 자체의 신호와 처리부를 가지는 독립적이고도 필요한 모든것을 다 갖춘 부분품으로 만든다.

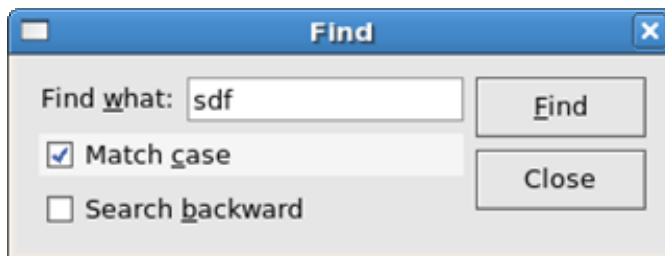



그림 2-1. Linux (KDE)에서 Find대화칸

원천코드는 두개의 파일 즉 finddialog.h와 finddialog.cpp에 보관된다. finddialog.h부터 보자.

```
1 #ifndef FINDDIALOG_H
2 #define FINDDIALOG_H
3 #include <qdialog.h>
4 class QCheckBox;
5 class QLabel;
6 class QLineEdit;
7 class QPushButton;
```

1행과 2행(그리고 27행)은 머리부파일이 여러번 포함되지 않도록 한다.

3행은 Qt에서 대화칸의 기초클라스인 QDialog의 정의를 포함한다. QDialog는 QWidget를 계승한다.

4∼7행은 대화칸을 실현하는데 사용하는 Qt클라스들의 앞방향선언이다. 앞방향선언(forward declaration)은 클라스정의(보통 자체의 머리부파일에 배치된다.)가 제공하는 자세한 내용을 모두 주지 않고도 클라스가 존재한다는것을 C++콤파일러에게 알려준다.

그다음 FindDialog를 QDialog의 파생클라스로 정의한다.

```
8 class FindDialog : public QDialog
9 {
10   Q_OBJECT
11 public:
12   FindDialog(QWidget *parent = 0, const char *name = 0);
```

클라스정의의 선두에 있는 Q_OBJECT마크로는 신호나 처리부들을 정의하는 모든 클라스들에서 반드시 필요하다.

FindDialog구성자는 Qt창문부품클라스들의 전형이다. parent파라메터는 부모창문부품을 지정하고 name파라메터는 창문부품에 이름을 준다. 이름은 임의로 선택할수 있고 주로 오유수정과 시험에 쓰인다.

```
13 signals:
14   void findNext(const QString &str, bool caseSensitive);
15   void findPrev(const QString &str, bool caseSensitive);
```

Signals부분에서는 사용자가 Find단추를 찰칵할 때 대화칸이 발생하는 두개의 신호를 선언한다. Search backward선택이 허용되면 대화칸은 findPrev()를 발생하고 그렇지 않으면 findNext()를 발생한다.

signals예약어는 사실 마크로이다. C++앞처리기가 signals를 표준 C++로 변환한 다음 콤파일러가 콤파일한다.

```
16 private slots:
17   void findClicked();
18   void enableFindButton(const QString &text);
19 private:
20   QLabel *label;
21   QLineEdit *lineEdit;
22   QCheckBox *caseCheckBox;
23   QCheckBox *backwardCheckBox;
24   QPushButton *findButton;
25   QPushButton *closeButton;
26 };
27 #endif
```

클라스의 private부분에서는 두개의 처리부를 선언한다. 처리부들을 실현하기 위해서는 대

화칸의 자식창문부품들의 대부분을 호출해야 하므로 그것들의 지적자들도 보관한다. slots예약어는 signals와 같은 마크로로서 C++콤파일러가 인식할수 있는 코드로 전개된다.

변수들이 모두 지적자로 선언되였고 머리부파일에서 변수들을 사용하지 않으므로 콤파일러는 완전한 클라스정의를 요구하지 않으며 앞방향선언이면 충분하다. 그대신에 관련한 머리부파일들 (<qcheckbox.h>, <qlabel.h> 등)을 포함하였지만 될수록 앞방향선언을 사용하면 콤파일속도가 좀 더 빨라진다.

그러면 FindDialog클라스의 실현을 포함하는 finddialog.cpp를 고찰하자.

```
1 #include <qcheckbox.h>
2 #include <qlabel.h>
3 #include <qlayout.h>
4 #include <qlineedit.h>
5 #include <qpushbutton.h>
6 #include "finddialog.h"
```

우선 finddialog.h와 함께 사용하려는 모든 Qt클라스들에 대한 머리부파일들을 포함한다. Qt클라스들에 대한 대부분의 머리부파일은 소문자로 된 클라스이름과 .h확장자를 가진다.

```
7 FindDialog::FindDialog(QWidget *parent, const char *name)
8 : QDialog(parent, name)
9 {
10   setCaption(tr("Find"));
11   label = new QLabel(tr("Find &what:"), this);
12   lineEdit = new QLineEdit(this);
13   label->setBuddy(lineEdit);
14   caseCheckBox = new QCheckBox(tr("Match &case"), this);
15   backwardCheckBox = new QCheckBox(tr("Search &backward"), this);
16   findButton = new QPushButton(tr("&Find"), this);
17   findButton->setDefault(true);
18   findButton->setEnabled(false);
19   closeButton = new QPushButton(tr("Close"), this);
```

8행에서는 parent와 name파라메터를 기초클라스구성자에 넘긴다.

10행에서는 창문의 제목을 Find로 설정한다. 문자렬주위의 tr()함수는 문자렬을 여러가지 언어로 번역할수 있게 한다. tr()함수는 Q_OBJECT마크로를 포함하는 QObject와 그 파생클라스에서 선언된다.

다음으로 자식창문부품들을 만든다. &기호를 사용하여 지름건을 지적한다. 례를 들면 16행은 사용자가 Alt+F를 누를 때 동작하는 Find단추를 만든다. 또한 &기호는 초점을 조종하는데 쓰인다. 11행에서는 지름건이 Alt+W인 표식자(label)을 만들고 13행에서는 표식자의 짝패를

행편집기(line editor)로 설정한다. 짝패(buddy)는 표식자의 지름건을 누를 때 초점을 받아들이는 창문부품이다. 그러므로 사용자가 Alt+W(표식자의 지름건)를 누를 때 초점은 행편집기(짝패)로 간다.

17행에서는 setDefault(true)를 호출함으로써 Find단추를 대화칸의 기정단추(default button)로 만든다. (Qt는 모든 가동환경에 대하여 TRUE와 FALSE를 제공하며 그것들을 표준 true와 false의 동의어로 사용한다. 그렇지만 true와 false를 유지하지 않는 낡은 콤파일러를 사용하는 경우에만 코드에서 대문자판을 사용할 필요가 있다.) 기정단추는 사용자가 Enter건을 누를 때 눌리우는 단추이다. 18행에서는 Find단추를 비능동상태로 한다. 창문부품이 비능동으로 될 때 보통 회색으로 표시되고 사용자가 조작할수 없다.

```
20  connect(lineEdit, SIGNAL(textChanged(const QString &)),
21          this, SLOT(enableFindButton(const QString &)));
22  connect(findButton, SIGNAL(clicked()),
23          this, SLOT(findClicked()));
24  connect(closeButton, SIGNAL(clicked()),
25          this, SLOT(close()));
```

비공개처리부 enableFindButton(const QString &)은 행편집기안의 본문이 달라질 때마다 호출된다. 비공개처리부 findClicked()는 사용자가 Find단추를 찰칵할 때 호출된다. 대화칸은 사용자가 Close를 찰칵할 때 닫긴다. close()처리부는 QWidget로부터 파생되고 그 기정동작은 창문부품을 숨기는것이다. enableFindButton()과 findClicked()처리부의 코드는 후에 고찰한다.

QObject는 FindDialog의 기초클라스이므로 connect()호출앞에서 QObject::앞붙이를 생략할수 있다.

```
26  QHBoxLayout *topLeftLayout = new QHBoxLayout;
27  topLeftLayout->addWidget(label);
28  topLeftLayout->addWidget(lineEdit);
29  QVBoxLayout *leftLayout = new QVBoxLayout;
30  leftLayout->addLayout(topLeftLayout);
31  leftLayout->addWidget(caseCheckBox);
32  leftLayout->addWidget(backwardCheckBox);
33  QVBoxLayout *rightLayout = new QVBoxLayout;
34  rightLayout->addWidget(findButton);
35  rightLayout->addWidget(closeButton);
36  rightLayout->addStretch(1);
37  QHBoxLayout *mainLayout = new QHBoxLayout(this);
38  mainLayout->setMargin(11);
39  mainLayout->setSpacing(6);
```

```
40  mainLayout->addLayout(leftLayout);
41  mainLayout->addLayout(rightLayout);
42 }
```

끝으로 배치관리자에 의하여 자식창문부품들을 배치한다. 배치관리자(layout manager)는 창문부품들의 크기와 위치를 관리하는 객체이다. Qt는 3가지 배치관리자를 제공한다. QHBoxLayout는 창문부품들을 수평으로 왼쪽에서 오른쪽으로(일부 문화어에서는 오른쪽에서 왼쪽으로) 배치하고 QVBoxLayout는 창문부품들을 수직으로 우에서 아래로 배치하며 QGridLayout는 창문부품들을 살창형태로 배치한다.

배치관리자는 창문부품들이나 다른 배치관리자들을 포함한다. QHBoxLayout와 QVBoxLayout, QGridLayout들을 여러가지 결합방식으로 배치하여 아주 복잡한 대화칸을 만들수 있다.

Find대화칸에서는 그림 2-2에 보여주는것처럼 2개의 QHBoxLayout와 두개의 QVBoxLayout를 사용한다. 바깥쪽 배치관리자가 기본배치(main layout)이고 FindDialog객체(this)를 그 부모로 하여 구성되며 대화칸의 전체령역을 차지하고 있다. 다른 3개 배치관리자는 보조배치(sub-layout)이다. 그림 2-2의 오른쪽아래에 있는 작은 《용수철》은 수축자항목(spacer item)이다. 수축자는 Find와 Close단추들아래의 빈 공간을 모두 리용하며 이 단추들이 배치관리자의 웃부분을 차지하도록 한다.

배치관리자클라스의 한가지 미묘한 점은 그것이 창문부품이 아니라는것이다. 그대신에 배치관리자클라스들은 QLayout를 계승하고 QLayout는 QObject를 계승한다. 그림에서 창문부품들은 실선륜곽선으로 표시하고 배치관리자들은 점선의 륜관선들로 표시하여 창문부품들과의 차이를 강조한다. 실행하는 응용프로그람에서는 배치관리자가 표시되지 않는다.

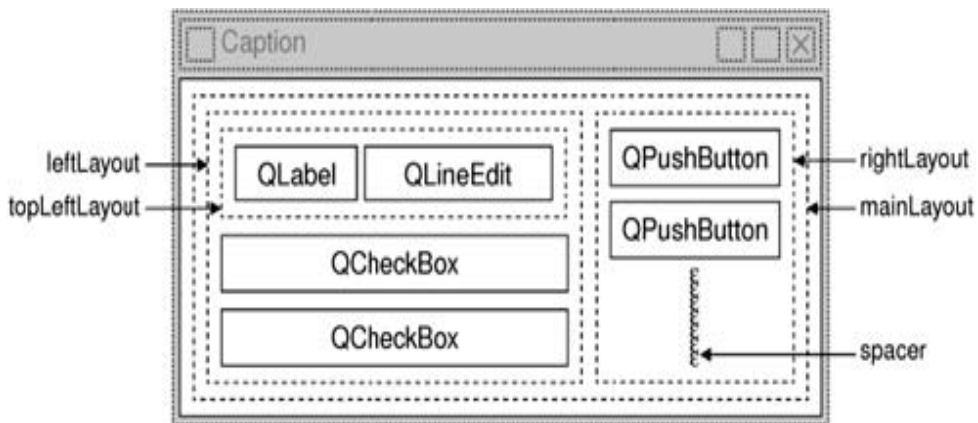



그림 2-2. Find대화칸의 배치관리자들

배치관리자가 창문부품은 아니지만 부모와 자식들을 가질수 있다. 배치에서 《부모》의 의미는 창문부품에서와 전혀 다르다. 배치관리자는 창문부품을 부모로 하여 구성되며 (mainLayout에서처럼) 배치관리자는 자동적으로 그자체를 창문부품우에 설치한다. 배치관리자가 부모없이 구성될 때 (topLeftLayout, leftLayout, rightLayout에서처럼) 그 배치관리자는

addLayout()에 의해 다른 배치관리자에 삽입되여야 한다.

Qt의 부모–자식기구(parent-child mechanism)는 QWidget와 QLayout의 기초클라스인 QObject에서 실현된다. 어떤 부모를 가지는 하나의 객체(창문부품, 배치관리자 혹은 다른 종류)를 창조할 때 부모는 그 객체를 자식들의 목록에 추가한다. 부모를 삭제할 때 자식목록을 순환하면서 매개 자식을 삭제한다. 자식들도 역시 스스로 자기 자식들을 모두 삭제하며 이 과정은 아무것도 남지 않을 때까지 재귀적으로 진행된다.

부모–자식기구는 기억관리를 훨씬 단순화하며 기억기가 루실될 위험을 줄인다. 명시적으로 삭제해야 할 객체들은 new에 의해 생성되며 부모가 없는 객체들이다. 그리고 부모를 삭제하기전에 자식객체를 삭제하면 Qt는 자동적으로 부모의 자식목록으로부터 그 객체를 삭제한다.

창문부품에서 부모는 한가지 의미를 더 가진다. 자식창문부품들은 부모의 령역안에 표시된다. 부모창문부품을 삭제할 때 자식이 기억기로부터 없어질뿐아니라 화면에서도 없어진다.

addLayout()에 의하여 어떤 배치관리자를 다른 배치관리자에 삽입하면 안쪽 배치관리자는 자동적으로 바깥쪽 배치관리자의 자식으로 되여 기억관리를 단순화한다. 이와 대조되게 addWidget()에 의해 창문부품을 배치관리자에 삽입할 때 그 창문부품은 부모를 변경하지 않는다.

그림 2-3은 창문부품과 배치관리자의 계승관계를 보여준다. 계승관계는 FindDialog구성자코드에서 new나 addLayout()호출을 포함하는 행들을 고찰하여 쉽게 끌어낼수 있다. 중요하게 알아두어야 할것은 배치관리자가 그것이 관리하는 창문부품들의 부모가 아니라는것이다.

```
FindDialog
 ├── QLabel (label)
 ├── QLineEdit (lineEdit)
 ├── QCheckBox (caseCheckBox)
 ├── QCheckBox (backwardCheckBox)
 ├── QPushButton (findButton)
 ├── QPushButton (closeButton)
 └── QHBoxLayout (mainLayout)
       ├── QVBoxLayout (leftLayout)
       │     └── QHBoxLayout (topLeftLayout)
       └── QVBoxLayout (rightLayout)
```

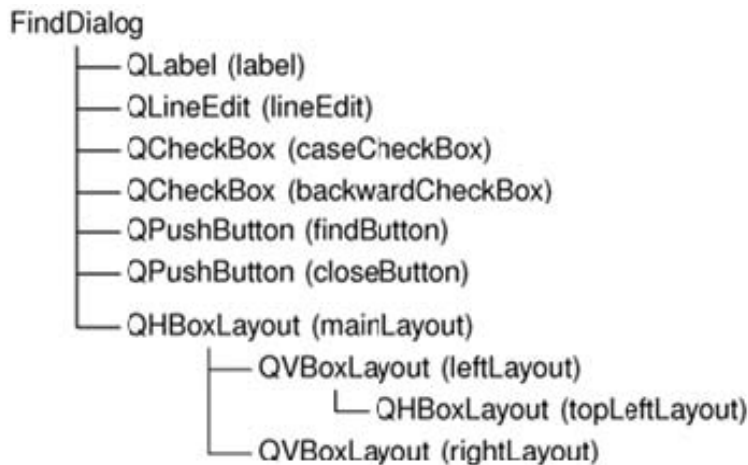



그림 2-3. Find대화칸의 부모–자식관계

배치관리자외에도 Qt는 배치관리자창문부품(layout widget) 즉 QHBox(제1장에서 사용), QVBox 및 QGrid를 제공한다. 이 클라스들은 자식창문부품들에 대하여 부모와 배치관리자로서 모두 봉사한다. 배치관리자창문부품들은 자그마한 실례들에서 배치관리자들보다 사용하기 편리하지만 유연성이 적고 더 많은 자원을 요구한다.

이로써 FindDialog의 구성자에 대한 개괄을 끝낸다. 대화칸의 창문부품과 배치관리자들

을 만드는데 new를 사용하므로 우리가 만든 매개 창문부품에 대하여 delete를 호출하는 해체자를 써야 하는것처럼 보인다. 그러나 그럴 필요는 없다 Qt는 자동적으로 부모가 해체될 때 자식객체들을 삭제하며 new에 의해 할당된 객체들은 모두 FindDialog의 자식들이다.

그러면 대화칸의 처리부에 대하여 고찰하자.

```
43 void FindDialog::findClicked()
44 {
45  QString text = lineEdit->text();
46  bool caseSensitive = caseCheckBox->isOn();
47  if (backwardCheckBox->isOn())
48       emit findPrev(text, caseSensitive);
49  else
50       emit findNext(text, caseSensitive);
51 }
52 void FindDialog::enableFindButton(const QString &text)
53 {
54  findButton->setEnabled(!text.isEmpty());
55 }
```

findClicked()처리부는 사용자가 Find단추를 찰칵할 때 호출된다. 이때 Search backward선택에 따라서 findPrev() 혹은 findNext()신호를 발생한다. emit예약어는 Qt에 고유한것으로서 다른 Qt확장기능처럼 C++앞처리기에 의해 표준 C++로 변환된다.

enableFindButton()처리부는 사용자가 행편집기안의 본문을 변경할 때마다 호출된다. 이 처리부는 편집기안에 본문이 있으면 단추를 허용하고 그렇지 않으면 금지한다.

이러한 2개의 처리부들로 대화칸을 완성한다. 이제는 main.cpp파일을 창조하여 FindDialog 창문부품을 시험해볼수 있다.

```
1 #include <qapplication.h>
2 #include "finddialog.h"
3 int main(int argc, char *argv[])
4 {
5  QApplication app(argc, argv);
6  FindDialog *dialog = new FindDialog;
7  app.setMainWidget(dialog);
8  dialog->show();
9  return app.exec();
10 }
```

보통 qmake를 실행하여 프로그람을 콤파일한다. FindDialog클라스정의에 Q_OBJECT마크로

를 포함하므로 qmake가 생성한 makefile에는 Qt의 메타객체콤파일러 moc를 실행하기 위한 특수규칙들이 포함되여있다.

moc가 정확히 작업하기 위해서는 머리부파일에 클라스정의를 넣어서 실현파일로부터 분리해야 한다. moc가 생성한 코드에는 이 머리부파일을 포함하고 자체의 규칙을 추가한다.

Q_OBJECT마크로를 사용하는 클라스들에 대하여 moc를 실행해야 한다. qmake가 자동적으로 makefile에 필요한 규칙들을 추가하므로 이것은 문제로 되지 않는다. 그러나 qmake에 의해 makefile을 다시 생성하는것을 잊어버리고 moc를 실행하지 않으면 련결프로그람은 일부 함수들이 선언되였는데 실현되지 않았다고 오유를 통보한다. GCC는 다음과 같이 경고를 내보낸다.

```
finddialog.o(.text+0x28): undefined reference to 'FindDialog::QPaintDevice virtual table'
```

Visual C++의 출력은 다음과 같이 시작한다.

```
finddialog.obj : error LNK2001: unresolved external symbol
"public:~virtual bool __thiscall FindDialog::qt_property(int, int,class QVariant *)"
```

이러한 통보가 발생하면 qmake를 다시 실행하여 makefile을 갱신한 다음 응용프로그람을 다시 건설한다.

그러면 프로그람을 실행하자. 지름건들인 Alt+W,Alt+C, Alt+B 그리고 Alt+F가 제대로 동작하는가를 확인한다. Tab를 눌러서 창문부품들을 순환해본다. 기정타브순서는 창문부품들을 창조한 순서이다. QWidget::setTabOrder()를 호출하여 이 순서를 변경할수 있다.

제3장에서는 실제의 응용프로그람에서 Find대화칸을 리용하며 findPrev()와 findNext()신호들을 처리부들에 련결한다.

제2절. 신호와 처리부

신호와 처리부기구(signal and slot mechanism)는 Qt프로그람작성의 기초이다. 이것은 응용프로그람작성자가 서로 모르는 객체들을 모두 결합할수 있도록 한다. 이미 신호와 처리부를 서로 련결하고 자체의 신호와 처리부를 선언하며 자체의 처리부들을 실현하고 자체의 신호들을 발생시키는 방법에 대하여 보았다. 여기서는 이것에 대하여 좀 더 구체적으로 고찰한다.

처리부는 보통의 C++성원함수와 거의 같다. 처리부는 가상함수일수 있고 재정의될수 있으며 public, protected 혹은 private일수 있고 다른 C++성원함수처럼 직접 호출될수도 있다. 차이는 처리부가 신호에 련결될수 있고 그러한 경우에 처리부는 신호가 발생될 때마다 자동적으로 호출된다는것이다.

connect()문은 다음과 같다.

connect (*sender*, SIGNAL(*signal*), *receiver*, SLOT(*slot*));

여기서 *sender*와 *receiver*는 QObject의 지적자들이고 *signal*과 *slot*는 파라메터이름이 없는 함수기호(signature)이다. SIGNAL()과 SLOT()마크로는 인수를 문자렬로 변환한다.

지금까지 본 실례들에서는 항상 서로 다른 처리부들에 서로 다른 신호들을 련결하였다. 또한 다음과 같은 특성들이 있다.

• 하나의 신호를 여러개의 처리부에 련결할수 있다.

```
connect(slider, SIGNAL(valueChanged(int)), spinBox, SLOT(setValue(int)));
connect(slider, SIGNAL(valueChanged(int)), this, SLOT(updateStatusBarIndicator(int)));
```

신호가 발생될 때 처리부들은 임의의 순서로 하나씩 호출된다.

• 여러개의 신호를 같은 처리부에 련결할수 있다.

```
connect(lcd, SIGNAL(overflow()), this, SLOT(handleMathError()));
connect(calculator, SIGNAL(divisionByZero()), this, SLOT(handleMathError()));
```

어느 한 신호가 발생될 때 그 처리부가 호출된다.

• 하나의 신호를 다른 신호에 련결할수 있다.

```
connect(lineEdit, SIGNAL(textChanged(const QString &)),this,
          SIGNAL(updateRecord(const QString &)));
```

첫째 신호가 발생될 때 둘째 신호도 역시 발생된다. 이것을 별도로 하여 신호–신호련결을 신호–처리부련결과 구별할수 없다.

• 련결을 삭제할수 있다.

```
disconnect(lcd, SIGNAL(overflow()),this, SLOT(handleMathError()));
```

Qt는 객체가 삭제될 때 그 객체를 포함하는 모든 련결을 자동적으로 삭제하므로 이것은 드물게 요구된다.

하나의 신호를 하나의 처리부나 다른 신호에 련결할 때 같은 형의 파라메터들이 같은 순서로 놓여야 한다.

```
connect(ftp, SIGNAL(rawCommandReply(int, const QString &)), this,
          SLOT(processReply(int, const QString &)));
```

하나의 신호가 거기에 련결되는 처리부보다 더 많은 파라메터를 가지면 나머지 파라메터들은 단순히 무시된다.

```
connect(ftp, SIGNAL(rawCommandReply(int, const QString &)), this,
          SLOT(checkErrorCode(int)));
```

파라메터형들이 호환되지 않거나 신호나 처리부가 존재하지 않으면 Qt는 실행시에 경고를 내보낸다. 마찬가지로 Qt는 신호나 처리부기호에 파라메터이름이 포함되면 경고를 내보낸다.

Qt의 메타객체체계

Qt의 주요한 성과의 하나는 임의의 부분품을 그것이 련결된 다른 부분품들에 대하여 전혀 모르고도 서로 결합할수 있는 독립적인 쏘프트웨어부분품들을 창조하는 기구에 의하여 C++를 확장한것이다.

그러한 기구를 메타객체체계(meta-object system)라고 부르며 두가지 주요한 기능 즉 신호 및 처리부와 자체검토(introspection)를 제공한다. 자체검토기능은 신호와 처리부를 실현하는데 필요되며 응용프로그람작성자들이 객체와 그 클라스이름에 의해 유지되는 신호와 처리

부의 목록을 비롯하여 QObject파생클라스들에 대한 《메타정보》를 실행시에 얻을수 있게 한다. 또한 이 기구는 Qt Designer용의 속성(property)과 국제화용본문번역(text translation)을 유지한다.

표준 C++는 Qt의 메타객체체계가 요구하는 동적메타정보에 대한 지원을 제공하지 않는다. Qt는 이 문제를 해결하기 위하여 moc라는 도구를 따로 제공한다. moc는 Q_OBJECT클라스정의들을 문법적으로 해석하고 그 정보를 C++함수들에서 사용할수 있게 만든다. moc가 순수 C++에 의해 그 기능을 모두 실현하므로 Qt의 메타객체체계는 임의의 C++콤파일러에서도 작업한다.

이 기구는 다음과 같이 작업한다.

- Q_OBJECT마크로는 QObject의 매개 파생클라스에서 실현되여야 하는 자체검토함수들 즉 metaObject()와 className(), tr() 등을 선언한다.
- Qt의 moc도구는 Q_OBJECT에 의해 선언된 함수들과 신호들의 실현을 생성한다.
- connect(), disconnect()와 같은 QObject성원함수들은 작업할 때 자체검토함수들을 리용한다.

이러한 모든것들은 qmake와 moc, QObject에 의하여 자동적으로 조종되므로 그것을 고찰할 필요가 있다. 그러나 호기심이 있다면 moc가 생성한 C++원천파일들을 보고 그 실현이 어떻게 작업하는가를 알수 있다.

지금까지는 창문부품과 함께 신호와 처리부만 리용하였다. 그러나 기구자체는 QObject에서 실현되며 GUI프로그람작성으로 제한되지 않는다. 이 기구는 임의의 QObject파생클라스들이 사용할수 있다.

```
class Employee : public QObject
{
    Q_OBJECT
public:
    Employee() { mySalary = 0;}
    int salary() const { return mySalary; }
public slots:
    void setSalary(int newSalary);
signals:
    void salaryChanged(int newSalary);
private:
    int mySalary;
};
void Employee::setSalary(int newSalary)
{
```

```
        if (newSalary != mySalary) {
            mySalary = newSalary;
            emit salaryChanged(mySalary);
        }
    }
```

setSalary()처리부를 실현하는 방법을 기억하자. newSalary != mySalary이면 오직 salaryChanged()신호만 발생한다. 이것은 주기적인 련결에 의해 무한순환이 일어나지 않도록 한다.

제3절. 대화칸의 고속설계

Qt는 재미있고 직관적으로 코드를 쓸수 있게 설계되였으며 순수 C++원천코드를 쓰는 방법으로 Qt응용프로그람들을 개발할수 있다. Qt Designer는 프로그람작성자들에게 유효한 선택들을 제공하며 시각적으로 설계한 폼을 원천코드와 결합할수 있게 한다.

이 절에서는 Qt Designer에 의하여 그림 2-4에 보여주는 Go-to-Cell대화칸을 창조한다. 이것을 코드로서 수행하든 Qt Designer에서 수행하든 대화칸의 창조는 항상 아래와 같은 기본단계를 가지게 된다.

· 자식창문부품들을 창조하고 초기화한다.
· 자식창문부품들을 배치관리자안에 놓는다.
· 타브순서를 설정한다.
· 신호–처리부련결들을 확립한다.
· 대화칸의 사용자정의처리부들을 실현한다.

Qt Designer를 기동하려면 Windows에서는 Start차림표에서 Qt 3.3.x|Qt Designer를 찰칵하고 Unix에서는 지령행에서 designer라고 입력하고 Mac OS X Finder에서는 designer를 두번 찰칵한다. Qt Designer가 기동할 때 형판들의 목록이 펼쳐진다. Dialog형판을 선택한 다음 OK를 찰칵한다. 그러면 Form1창문이 나타난다.

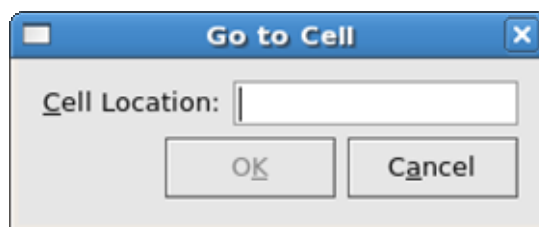



그림 2-4. Go-to-Cell대화칸

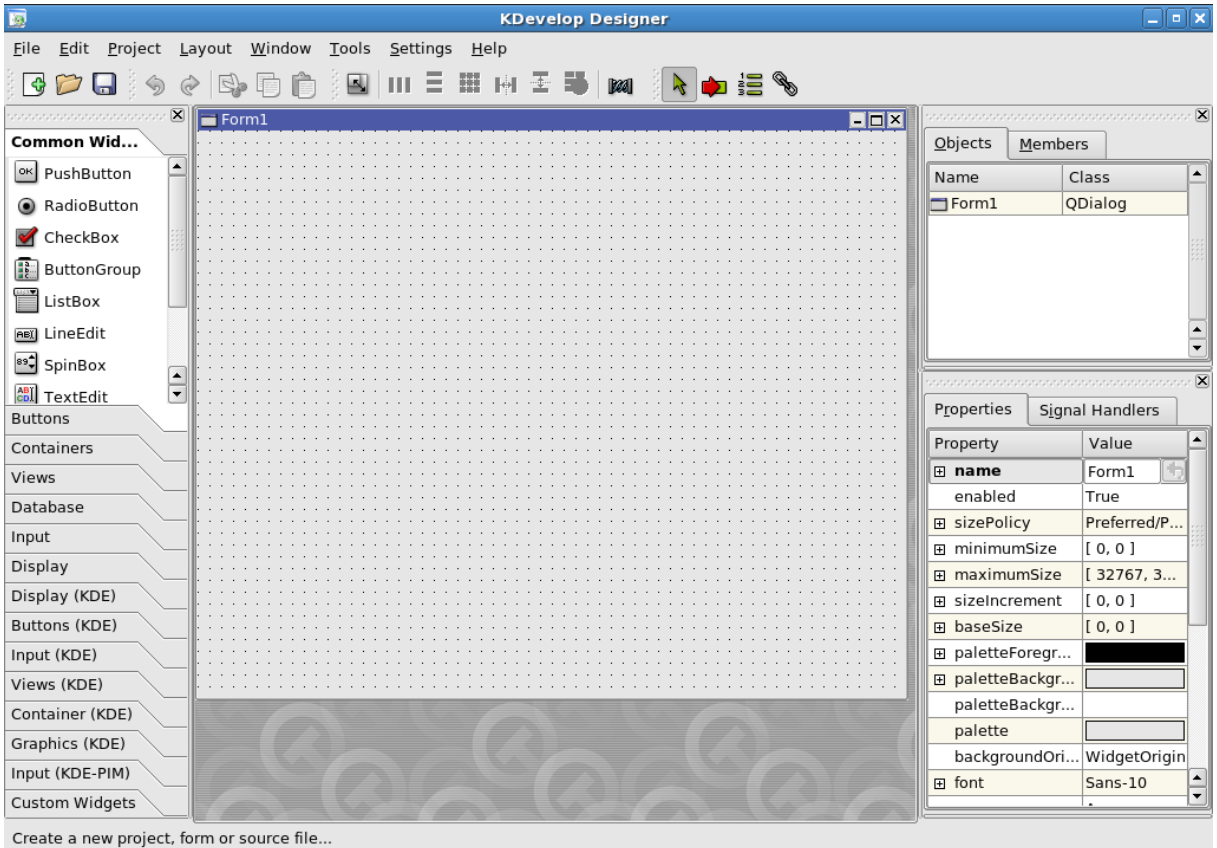



그림 2-5. 빈 폼을 가지는 Qt Designer

첫 단계에서는 자식창문부품들을 창조하여 폼우에 배치한다. 본문표식자(text label) 하나, 행편집기(line editor) 하나, (수평)수축자(spacer) 하나 그리고 두개의 누름단추(pushbutton)들을 창조한다. 매개 항목에 대하여 Qt Designer의 기본창문의 왼쪽에 있는 toolbox안의 이름이나 그림기호를 찰칵하고 그 항목을 배치하려는 폼의 위치를 찰칵한다. 그리고 항목의 길이가 더 짧아질 때까지 폼의 바닥을 끌고간다. 이리하여 그림 2-6과 비슷한 폼을 만들어낸다. 폼우에서 항목들의 위치를 지정하는데 너무 시간을 소비할 필요는 없다. 후에 Qt의 배치관리자들이 그것들을 정확히 배치한다.

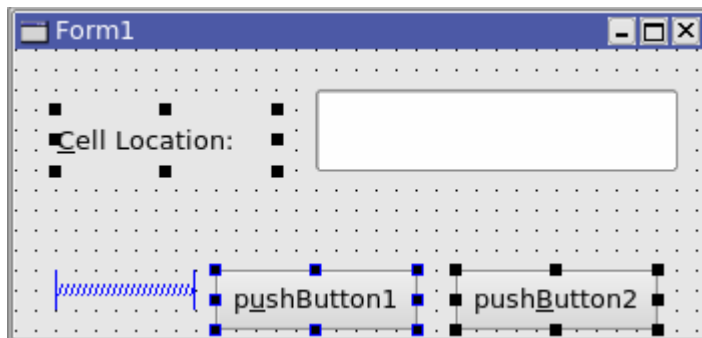



그림 2-6. 창문부품들을 가지는 폼

수축자항목은 Qt Designer에 푸른색 용수철로 표시된다. 수축자는 최종폼에 보이지 않는

다.

Qt Designer의 기본창문 오른쪽에 있는 속성편집기에 의하여 각 창문부품의 속성들을 설정한다.

① 본문표식자를 찰칵하고 name속성을 label로, text속성을 "&Cell Location:"로 설정한다.

② 행편집기를 찰칵하고 name속성을 lineEdit로 설정한다.

③ 수축자를 찰칵하고 수축자의 orientation속성이 Horizontal로 설정되였는가를 확인한다.

④ 첫째 단추를 찰칵하고 name속성을 okButton으로, enabled속성을 False로, default속성을 True로, text속성을 OK로 설정한다.

⑤ 둘째 단추를 찰칵하고 name속성을 cancelButton으로, text속성을 Cancel로 설정한다.

⑥ 폼의 배경을 찰칵하여 폼자체를 선택하고 name속성을 GoToCellDialog로, caption속성을 "Go to Cell"로 설정한다.

"&Cell Location"을 표시하는 본문표식자를 제외한 모든 창문부품들이 잘 보인다. Tools|Set Buddy를 찰칵한다. 표식자를 누른 상태에서 마우스를 행편집기까지 끌고가서 놓는다. 그러면 행편집기는 표식자의 짝패로 된다. 표식자의 buddy속성이 lineEdit로 설정되였는가를 검사하여 이것을 확인할수 있다.

그림 2-7. 속성모임을 가지는 폼

다음 단계에서는 폼우에 있는 창문부품들을 배치한다.

① Cell Location표식자를 찰칵하고 Shift를 누른 상태에서 표식자옆에 있는 행편집기를 찰칵하여 그것들을 둘다 선택한다. Layout|Lay Out Horizontally를 찰칵한다.

② 수축자를 찰칵한 다음 Shift를 누르면서 폼의 OK와 Cancel단추들을 찰칵한다. Layout|Lay Out Horizontally를 찰칵한다.

③ 폼의 배경을 찰칵하여 항목들의 선택을 해제한 다음 Layout|Lay Out Vertically을 찰칵한다.

④ Layout|Adjust Size를 찰칵하여 폼의 크기를 최량크기로 조절한다.

폼우에 나타나는 붉은선들은 창조된 배치관리자들을 보여준다. 배치관리자들은 폼이 실행될 때 절대로 나타나지 않는다.

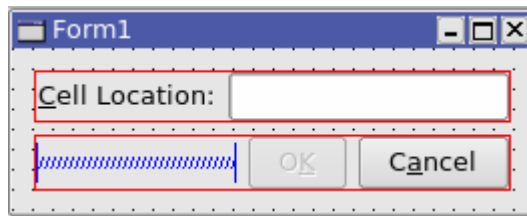



그림 2-8. 배치관리자들을 가지는 폼

이제 Tools|Tab Order를 찰칵한다. 초점을 받아들이는 매개 창문부품옆에 있는 푸른색 원안에 수자가 나타난다. 초점을 받아들이려는 순서로 매개 창문부품을 찰칵한 다음 Esc를 누른다.

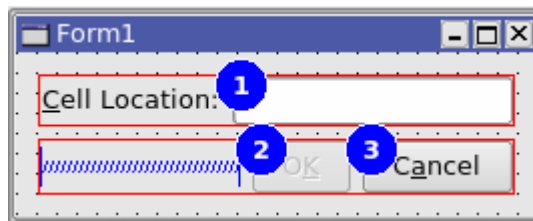



그림 2-9. 폼의 타브순서 설정

이렇게 하면 폼설계는 끝난다. 신호-처리부련결들을 설정하고 사용자정의처리부들을 실현하여 폼이 동작하게 하는 부분만이 남는다. Edit|Connections를 찰칵하여 련결편집기를 펼친다.

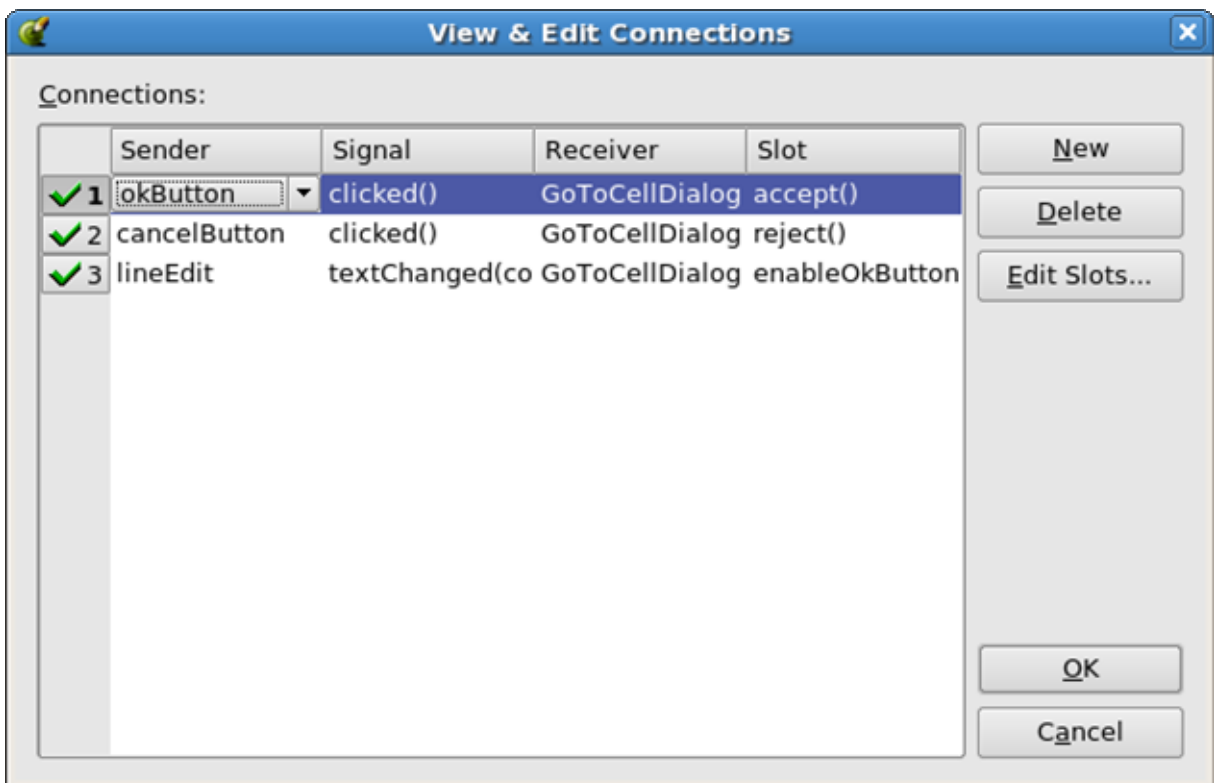



그림 2-10. Qt Designer의 련결(련결들을 만든후)

3개의 련결을 만든다. 련결을 창조하려면 New를 찰칵하고 내리펼침복합칸들을 리용하여 Sender와 Signal, Receiver, Slot마당들을 설정한다.

okButton의 clicked()신호를 GoToCellDialog의 accept()처리부에 련결한다. cancelButton의 clicked()신호를 GoToCellDialog의 reject()처리부에 련결한다. Edit Slots를 찰칵하여 Qt Designer의 처리부편집기(그림 2-11)를 펼치고 enableOkButton()비공개처리부를 창조한다. 끝으로 lineEdit의 textChanged(const QString &)신호를 GoToCellDialog의 새로운 enableOkButton()처리부에 련결한다.

대화칸을 미리 보려면 Preview|Preview Form차림표선택을 찰칵한다. Tab를 반복하여 눌러서 타브순서를 검사한다. Alt+C를 눌러서 초점을 행편집기로 옮긴다. Cancel을 찰칵하여 대화칸을 닫는다.

대화칸을 gotocell이라는 등록부에 gotocelldialog.ui로서 보관하고 일반본문편집기에 의하여 같은 등록부에 main.cpp파일을 창조한다.

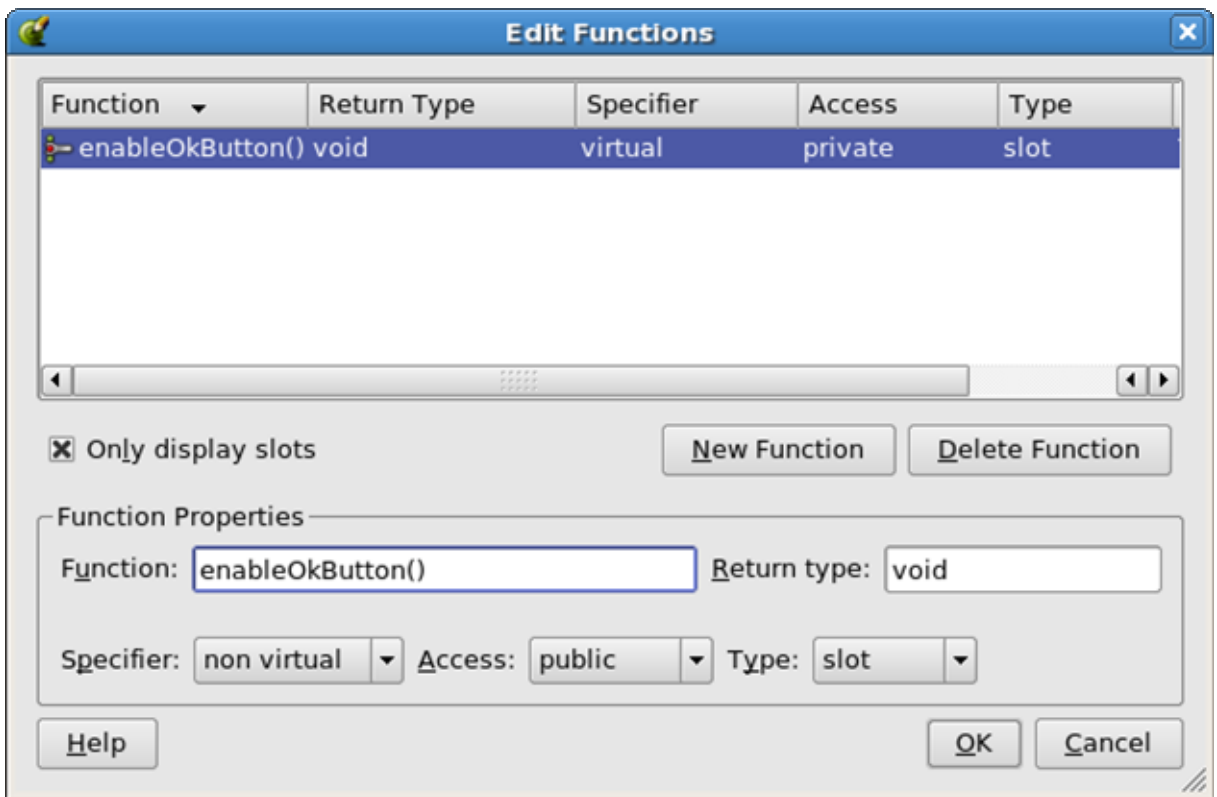



그림 2-11. Qt Designer의 처리부편집기

```
#include <qapplication.h>
#include "gotocelldialog.h"
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
```

```
        GoToCellDialog *dialog = new GoToCellDialog;
        app.setMainWidget(dialog);
        dialog->show();
        return app.exec();
    }
```

이제 qmake를 실행하여 .pro파일과 makefile을 창조한다. (qmake -project; qmake gotocell.pro). qmake도구는 사용자대면부파일 gotocelldialog.ui를 탐색하여 적당한 makefile규칙들을 생성하고 gotocelldialog.h와 gotocelldialog.cpp를 창조하는 고급한 도구이다. .ui파일은 Qt의 사용자대면부 콤파일러uic에 의해 C++로 변환된다.

Qt Designer리용의 한가지 우점은 원천코드를 혼돈시키지 않고 프로그람작성자들이 폼설계를 수정할 자유를 얻는다는데 있다. C++코드를 쓰는 방법으로 순수 폼을 개발할 때 설계에 대한 변경은 시간을 소비하게 한다. Qt Designer를 리용할 때 uic는 변경된 폼들에 대해서만 원천코드를 재생하므로 시간을 랑비하지 않는다.

이제 프로그람을 실행하면 대화칸은 동작하지만 요구대로 정확히 기능하지 않는다.

· OK단추는 항상 비능동상태로 된다.

· 행편집기는 유효세포위치들만 받아들일 대신에 임의의 본문이나 다 받아들인다.

그러므로 이 문제를 해결하기 위한 코드를 써야 한다.

폼의 배경을 두번 련속 찰칵하여 Qt Designer의 코드편집기를 펼친다. 편집기창문에서 다음의 코드를 입력한다.

```
#include <qvalidator.h>
void GoToCellDialog::init()
{
    QRegExp regExp("[A–Za–z] [1–9] [0–9] {0, 2}");
    lineEdit->setValidator(new QRegExpValidator(regExp, this));
}
void GoToCellDialog::enableOkButton()
{
    okButton->setEnabled(lineEdit->hasAcceptableInput());
}
```

init()함수가 폼구성자(uic가 생성)의 끝에서 자동적으로 호출된다. 입력범위를 제한하기 위하여 유효기를 설정한다. Qt는 3개의 기본유효성확인클라스 즉 QIntValidator와 QDoubleValidator, QRegExpValidator를 제공한다. 여기서는 정규식 "[A-Za-z][1-9][0-9]{0,2}"을 가지고 QRegExpValidator를 리용한다. 이것은 하나의 영문자와 그뒤에 1～999까지의 수값으로 이루어진 값만이 유효하다는것을 의미한다.(정규식에 대해서는 QRegExp클라스방조를 참고.)

이것을 QRegExpValidator구성자에 넘김으로써 GoToCellDialog객체의 자식으로 만든다. 그

리하여 후에 QRegExpValidator의 부모가 삭제될 때 자동적으로 삭제된다.

enableOkButton()처리부는 행편집기가 유효세포위치를 포함하는가에 따라서 *OK*단추를 허용 혹은 금지한다. QLineEdit::hasAcceptableInput()는 init()함수에서 설정한 유효기를 사용한다.

```
GoToCellDialog
#include <qvalidator.h>

void GoToCellDialog::init()
{
    QRegExp regExp("[A-Za-z][1-9][0-9]{0,2}");
    lineEdit->setValidator(new QRegExpValidator(regExp, this));
}

void GoToCellDialog::enableOkButton()
{
    okButton->setEnabled(lineEdit->hasAcceptableInput());
}
Line: 13 Col: 1
```

그림 2-12. Qt Designer의 코드편집기

코드를 입력한 후에 다시 대화칸을 보관한다. 이때 2개 파일 즉 사용자대면부파일 gotocelldialog.ui와 C++원천파일 gotocelldialog.ui.h를 보관한다. 다시 응용프로그람을 구축하고 실행한다. 행편집기에 "A12"라고 입력하면 OK단추가 능동상태로 되는것을 알수 있다. 임의의 본문을 입력하여 유효기의 동작을 고찰한다. Cancel을 찰칵하여 대화칸을 닫는다.

이 실례에서는 Qt Designer에서 대화칸을 편집하고 Qt Designer의 코드편집기에 의하여 코드를 추가하였다. 대화칸의 사용자대면부는 .ui파일(XML기초파일형식)에 보관되고 코드는 .ui.h파일(C++원천파일)에 보관된다.

.ui.h의 다른 수법은 보통과 같이 Qt Designer로 .ui파일들을 창조하고 uic가 생성한 클라스를 계승하는 파생클라스를 창조하고 거기에 나머지 기능을 추가하는것이다. 례를 들면 Go-to-Cell대화칸에서 이것은 GoToCellDialog를 계승하는 GoToCellDialogImpl클라스를 창조하여 필요한 기능을 추가적으로 실현한다는것을 의미한다. 이 수법으로 .ui.h코드를 변환하는것은 간단하다. 결과는 다음의 머리부파일과 같다.

```
#ifndef GOTOCELLDIALOGIMPL_H
#define GOTOCELLDIALOGIMPL_H
#include "gotocelldialog.h"
class GoToCellDialogImpl : public GoToCellDialog
{
 Q_OBJECT
public:
    GoToCellDialogImpl(QWidget *parent = 0, const char *name = 0);
```

```
private slots:
    void enableOkButton();
};
#endif
```

그리고 원천파일은 다음과 같다.

```
#include <qlineedit.h>
#include <qpushbutton.h>
#include <qvalidator.h>
#include "gotocelldialogimpl.h"
GoToCellDialogImpl::GoToCellDialogImpl(QWidget *parent, const char *name)
     : GoToCellDialog(parent, name)
{
    QRegExp regExp("[A-Za-z][1-9][0-9]{0,2}");
    lineEdit->setValidator(new QRegExpValidator(regExp, this));
}
void GoToCellDialogImpl::enableOkButton()
{
    okButton->setEnabled(lineEdit->hasAcceptableInput());
}
```

파생클라스작성수법을 좋아하는 개발자들은 기초클라스 GoToCellDialogBase과 그의 모든 기능을 포함하면서 좋은 이름을 가지는 파생클라스 GoToCellDialog를 호출한다.

uic도구는 Qt Designer로 창조한 폼들에 기초한 파생클라스의 창조를 단순화하기 위한 지령행선택들을 제공한다. –subdecl를 사용하여 골격머리부파일을 생성하고 –subimpl을 사용하여 그와 짝을 이루는 실현파일을 생성한다.

제4절. 형태가 변하는 대화칸

앞에서는 고정된 창문부품만이 현시되는 대화칸창조방법을 보았다. 일부 경우에 형태를 변경할수 있는 대화칸을 제공하여야 한다. 형태가 변하는 대화칸에서 2가지의 가장 일반적인 종류가 확장대화칸와 여러페지대화칸이다. 두가지 대화칸은 Qt에서 순수 코드로 혹은 Qt Designer를 사용하여 실현할수 있다.

보통 확장대화칸은 간단한 형태로 표시되지만 대화칸의 단순한 형태와 확장된 형태사이를 사용자가 절환하게 하는 절환(toggle)단추를 가지고있다. 확장대화칸은 보통 림시사용자와 능력있는 사용자들의 요구를 만족시키려고 하는 응용프로그람들에 사용되며 사용자가 구체적인 선택을 알려고 정확히 요구하지 않는한 그 선택들을 숨긴다. 이 절에서는 Qt Designer에 의하여 그림 2-13에 보여주는 확장대화칸을 창조한다.

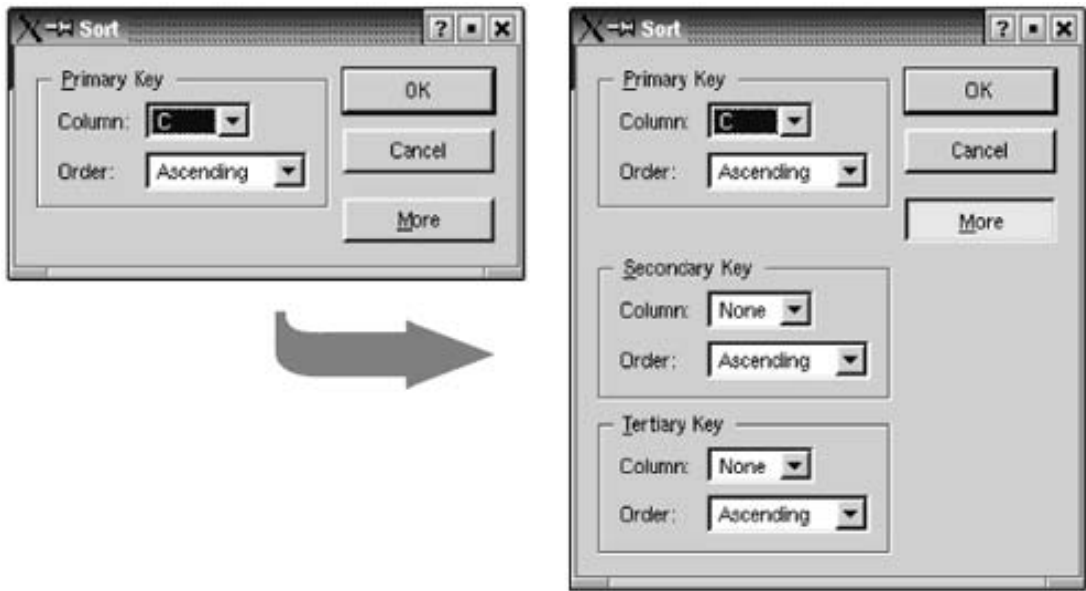



그림 2-13. 단순한 Sort대화칸와 확장된 Sort대화칸

이 대화칸은 표계산프로그람의 Sort대화칸이다. 여기서 사용자는 정렬하려는 하나이상의 렬들을 선택할수 있다. 단순한 대화칸은 사용자가 단일한 정렬건을 입력하게 하고 확장된 대화칸은 2개의 여분의 정렬건을 제공한다. More단추는 사용자가 단순한 형태와 확장형태사이를 절환하게 한다.

Qt Designer로 확장형태를 가지는 창문부품들을 창조하고 실행시에 필요할 때 둘째건과 셋째건을 숨긴다. 창문부품의 창조가 복잡한것같지만 그것은 Qt Designer로 간단히 수행할수 있다. 요점은 우선 첫째그룹을 창조하고 다음에 그것을 두번 복사하고 붙이기하여 둘째그룹과 셋째그룹을 얻는것이다.

① 그룹칸 하나, 본문표식자 두개, 복합칸 두개 그리고 수평수축자 하나를 창조한다.

② 그룹칸의 오른쪽아래구석을 끌어서 늘인다.

③ 다른 창문부품들을 그룹칸으로 옮기고 그것들을 그림 2-14(ㄱ)에 보여주는것과 근사하게 배치한다.

④ 둘째 복합칸의 오른변을 끌어서 첫 복합칸폭의 약 두배로 늘인다.

⑤ 그룹칸의 title속성을 "&Primary Key"로, 첫째 표식자의 text속성을 "Column:"으로, 둘째 표식자의 text속성을 "Order:"로 설정한다.

⑥ 첫째 복합칸을 두번 찰칵하여 Qt Designer의 목록칸편집기를 펼치고 본문 "None"을 가지는 항목을 하나 창조한다.

⑦ 둘째 복합칸을 두번 찰칵하고 Ascending항목과 Descending항목을 창조한다.

⑧ 그룹칸을 찰칵한 다음 Layout|Lay Out in a Grid를 찰칵한다. 이것은 그림 2-14(ㄴ)에 보여주는 배치관리자를 생성한다.

배치가 제대로 되지 않거나 오유가 발생하면 늘 Edit|Undo를 찰칵한 다음 배치할 창문부

품들의 위치를 다시 조절하고 다시 배치를 시도한다.

이제는 Secondary Key와 Tertiary Key그룹칸들을 추가한다.

① 충분히 여유를 주어 대화창문을 크게 만든다. 그룹칸을 선택하고 Edit|Copy를 찰칵한 다음 Edit|Paste를 찰칵하여 두개의 그룹칸을 추가한다. 두개의 새로운 그룹칸을 끌어서 그것들이 차지하여야 할 대략적인 위치로 가져간다. 그것들의 title속성을 각각 "&Secondary Key"와 "&Tertiary Key"로 변경한다.

ㄱ)정리없음　　ㄴ)정리있음

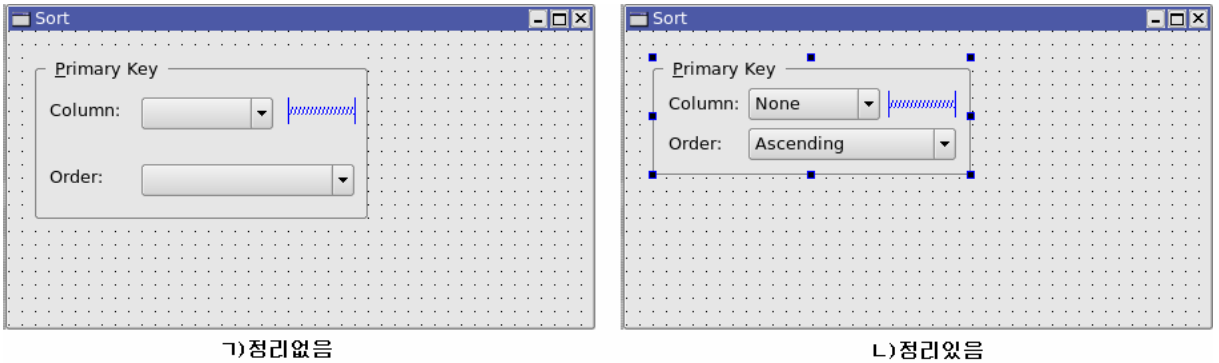



그림 2-14. 그룹칸의 자식들을 살창형태로 배치하기

② OK, Cancel 및 More단추들을 창조한다.

③ OK단추의 default속성을 True, More단추의 toggle속성을 True로 설정한다.

④ 두개의 수직수축자를 창조한다.

⑤ OK, Cancel 및 More단추들을 수직으로 배렬하고 Cancel과 More단추사이에 수직수축자를 배치한다. 그다음 4개의 항목을 모두 선택하고 Layout|Lay Out Vertically을 찰칵한다.

⑥ 주열쇠그룹칸과 둘째건그룹칸사이에 두번째 수직수축자를 배치한다.

⑦ 두개의 수직수축자항목들의 sizeHint속성을 (20, 10)로 설정한다.

⑧ 창문부품들을 그림 2-15(ㄱ)에 보여주는 살창형식으로 배렬한다.

⑨ Layout|Lay Out in a Grid를 찰칵한다. 이제는 폼이 그림 2-15(ㄴ)와 일치된다.

ㄱ)정리없음 ㄴ)정리있음

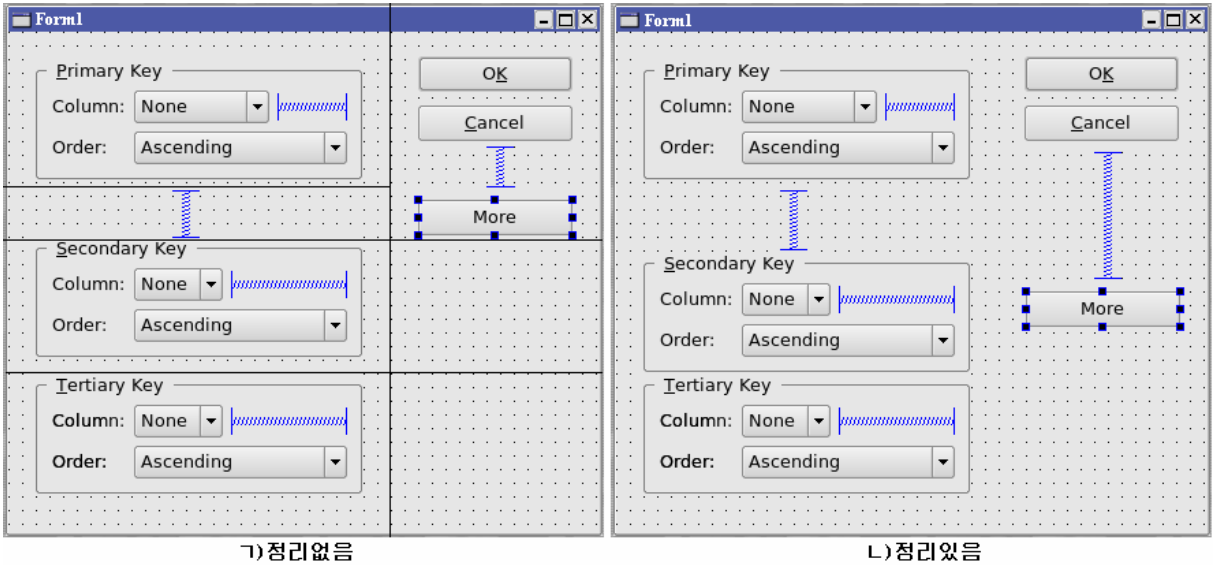



그림 2-15. 폼의 자식들을 살창형태로 배치하기

이리하여 살창배치관리자는 2개의 렬과 4개의 행, 총 8개의 세포를 제공한다. Primary Key 그룹칸, 제일 왼쪽의 수직수축자항목, Secondary Key그룹칸, 및 Tertiary Key그룹칸은 각각 한개 세포씩 차지한다. OK, Cancel 및 More단추들을 포함하는 수직배치관리자는 두개의 세포를 차지한다. 대화칸의 오른쪽 아래에 두개의 빈 세포가 남아있다. 이것이 마음에 들지 않으면 배치를 취소하고 창문부품들을 재배치하고 다시 배치한다.

폼의 resizeMode속성을 Auto로부터 Fixed로 변경하여 대화칸의 크기를 사용자가 조절할수 없게 만든다. 그다음 배치관리자는 크기조절에 대한 응답능력을 넘겨받으며 대화칸은 자식창문부품들이 표시되거나 은폐될 때 크기가 자동조절되여 대화칸이 늘 최량크기로 표시되도록 한다.

폼의 이름을 SortDialog로, 그 제목을 Sort로 변경한다. 자식창문부품들의 이름을 그림 2-16에 보여주는것처럼 설정한다.

끝으로 련결들을 설정한다.

① okButton의 clicked()신호를 폼의 accept()처리부에 련결한다.

② cancelButton의 clicked()신호를 폼의 reject()처리부에 련결한다.

③ moreButton의 toggled(bool)신호를 secondaryGroupBox의 setShown(bool)처리부에 련결한다.

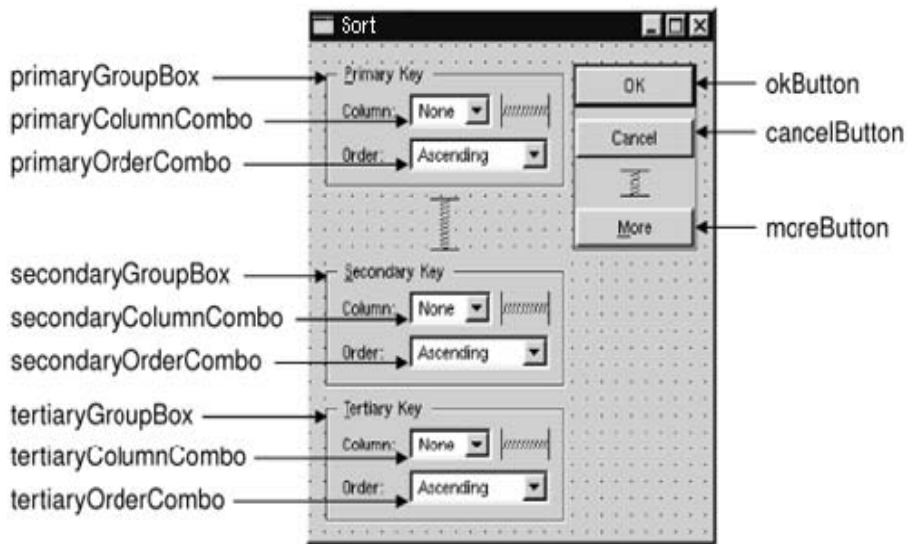



그림 2-16. 폼의 창문부품이름

④ moreButton의 toggled(bool)신호를 tertiaryGroupBox의 setShown(bool)처리부에 련결한다.

폼을 두번 련속 찰칵하여 Qt Designer의 C++코드편집기를 펼치고 다음의 코드를 입력한다.

```
1 void SortDialog::init()
2 {
3   secondaryGroupBox->hide();
4   tertiaryGroupBox->hide();
5   setColumnRange('A', 'Z');
6 }
7 void SortDialog::setColumnRange(QChar first, QChar last)
8 {
9   primaryColumnCombo->clear();
10  secondaryColumnCombo->clear();
11  tertiaryColumnCombo->clear();
12  secondaryColumnCombo->insertItem(tr("None"));
13  tertiaryColumnCombo->insertItem(tr("None"));
14  primaryColumnCombo->setMinimumSize(
15  secondaryColumnCombo->sizeHint());
16  QChar ch = first;
17  while (ch <= last) {
18      primaryColumnCombo->insertItem(ch);
19      secondaryColumnCombo->insertItem(ch);
```

```
20          tertiaryColumnCombo->insertItem(ch);
21          ch = ch.unicode() + 1;
22    }
23 }
```

init()함수는 대화칸의 두번째와 세번째그룹부분을 은페한다.

setColumnRange()처리부는 표에서 선택한 렬에 기초하여 복합칸들의 내용을 초기화한다. 두번째와 세번째그룹의 복합칸에 None항목을 삽입한다. Qt Designer의 처리부편집기를 리용하여 이 처리부를 창조하지 않았어도 Qt Designer는 코드에서 새로운 처리부가 창조된것을 탐지하고 uic는 자동적으로 SortDialog클라스정의에 정확한 함수선언을 생성한다.

14행과 15행은 배치형식을 보여준다. QWidget::sizeHint()함수는 배치체계가 받아들이는 창문부품의 《리상적인》 크기를 돌려준다. 이것은 배치체계에 의하여 서로 다른 종류의 창문부품들이나 서로 다른 내용을 가지는 류사한 창문부품들에 각이한 크기를 할당하는 기초로 된다. 즉 복합칸들에서 이것은 None을 포함하는 둘째와 셋째 복합칸들이 한개 문자만을 포함하는 첫째 복합칸보다 커진다는것을 의미한다. 이러한 비일관성을 피하기 위하여 첫째 복합칸의 최소크기를 둘째 복합칸의 리상적인 크기로 설정한다.

여기서 main()은 'C'～'F'렬을 포함하도록 범위를 설정한 다음 대화칸을 표시하는 기능을 시험한다.

```
#include <qapplication.h>
#include "sortdialog.h"
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    SortDialog *dialog = new SortDialog;
    app.setMainWidget(dialog);
    dialog->setColumnRange('C', 'F');
    dialog->show();
    return app.exec();
}
```

이로서 확장대화칸을 완성한다. 실례에서 설명하는것처럼 확장대화칸은 일반대화칸보다 설계하는것이 그리 힘들지 않다. 필요한것은 절환단추 하나와 여러개의 신호–처리부련결 그리고 크기조절할수 없는 하나의 배치관리자이다.

형태가 변하는 다른 일반대화칸으로서 여러페지대화칸은 Qt에서 코드로 혹은 Qt Designer에 의해 실현하는것이 훨씬 간단하다. 여러페지대화칸은 여러가지 방법으로 만들수 있다.

· QTabWidget는 독립적으로 쓰일수 있다. 이것은 웃끝에 내부QWidgetStack를 조종하는 타브띠를 제공한다.

· QListBox와 QWidgetStack는 함께 사용할수 있으며 이때 QListBox의 현재항목을 리용하여 QWidgetStack가 표시할 페지를 결정한다.

· QListView 혹은 QIconView는 QListBox와 비슷한 방법으로 QWidgetStack에서 사용될수 있다.

QWidgetStack클라스는 《6장. 배치관리》에서 설명한다.

제5절. 동적대화칸

동적대화칸은 실행시에 Qt Designer의 .ui파일로부터 창조되는 대화칸이다. 동적대화칸은 uic에 의해 C++코드로 변환되지 않는다. 그대신에 .ui파일은 다음과 같은 방법으로 실행시에 QWidgetFactory클라스에 의하여 적재된다.

```
QDialog *sortDialog = (QDialog *)QWidgetFactory::create("sortdialog.ui");
```

QObject::child()에 의하여 폼의 자식창문부품들을 호출할수 있다.

```
QComboBox *primaryColumnCombo = (QComboBox *)sortDialog->
                    child("primaryColumnCombo", "QComboBox");
```

child()함수는 대화칸이 주어진 이름과 형이 일치하는 자식을 가지지 않으면 null지적자를 돌려준다.

QWidgetFactory클라스는 개별적인 서고에 배치된다. Qt응용프로그람으로부터 QWidgetFactory를 사용하려면 다음 행을 응용프로그람의 .pro파일에 추가해야 한다.

```
LIBS += -lqui
```

이 문법은 모든 가동환경에서 적용할수 있다.

동적대화칸은 응용프로그람을 다시 콤파일하지 않고도 폼의 배치를 변경할수 있게 한다.

제6절. 기본창문부품과 대화칸클라스

Qt는 대부분의 상황에 적합한 기본창문부품과 공통대화칸들을 제공한다. 이 절에서는 그것들에 대하여 소개한다. 일부 전문화된 창문부품들은 후에 언급한다. QMenuBar, QPopupMenu, QToolBar와 같은 기본창문창문부품들은 제3장에서 설명하고 QDataView, QDataTable과 같은 자료기지창문부품들은 제12장에서 설명한다. 대부분의 기본창문부품과 대화칸은 이 책에서 제시하는 실례들에서 사용된다. 아래의 그림들에서 창문부품들은 전형적인 Windows형식으로 보여준다.

그림 2-17. Qt의 단추창문부품들

Qt는 세 종류의 단추 즉 QPushButton, QCheckBox, QRadioButton을 제공한다. QPushButton은

찰칵할 때 작용을 초기화하는데 제일 많이 쓰이지만 절환단추와 같이 동작할수도 있다. QRadioButton은 보통 QButtonGroup안에서 서로 배타적인 선택에 쓰이고 QCheckBox는 독립적인 on/off선택에 쓰인다.

Qt의 용기창문부품은 다른 창문부품들을 포함하는 창문부품이다. 또한 QFrame은 단순히 그 자체에 직선을 그리는데 쓰이며 다른 수많은 창문부품클라스 특히 QLabel과 QLineEdit이 그것을 계승한다. QButtonGroup는 표시되지 않으며 시각적으로 QGroupBox와 등가하다.

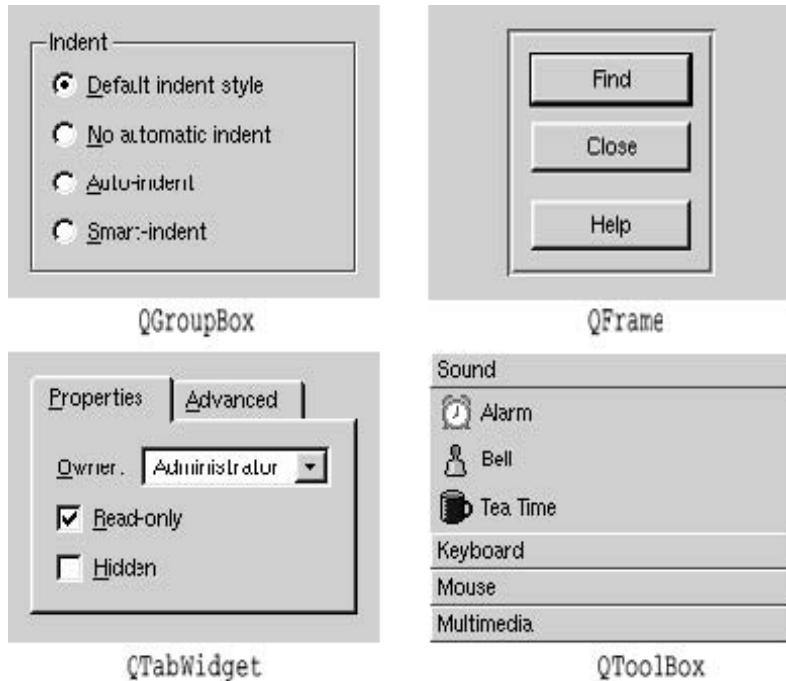



그림 2-18. Qt의 용기창문부품들

QTabWidget와 QToolBox는 여러페지창문부품이다. 매개 페지는 하나의 자식창문부품이고 페지들에는 0부터 번호가 붙는다.

QListBox QListView

QIconView QTable

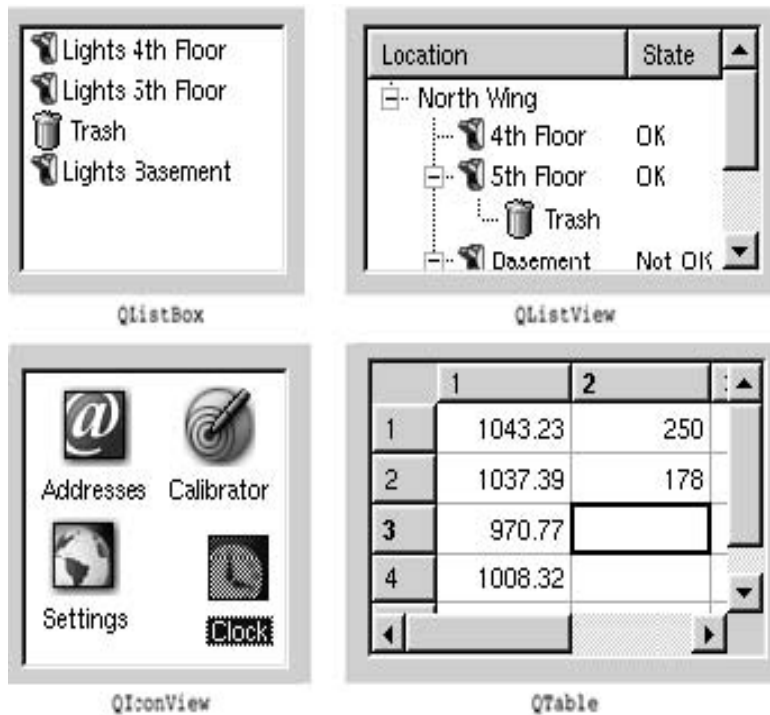



그림 2-19. Qt의 항목보기창문부품들

항목보기는 대량자료를 조종할수 있도록 최적화되며 흔히 흘림띠를 사용한다. 흘림띠기구는 항목보기와 다른 종류의 보기들의 기초클라스인 QScrollView에 의해 실현된다.

QLabel QLCDNumber QProgressBar

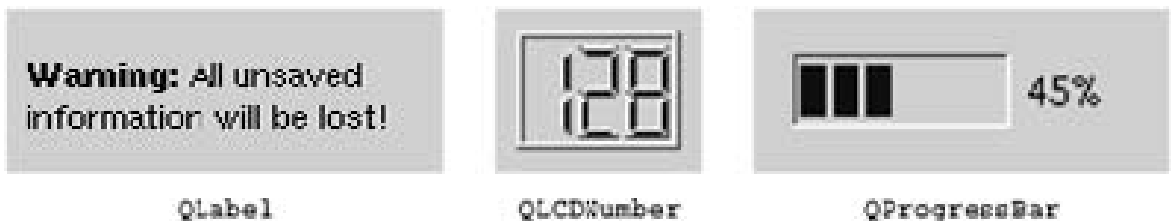



그림 2-20. Qt의 현시창문부품들

Qt는 순수 정보를 현시하는데 쓰이는 창문부품들을 제공한다. QLabel은 그중에서 가장 중요하며 리치본문(단순한 HTML형식의 문법을 리용)과 화상을 표시하는데 쓰인다.

QTextBrowser(표시되지 않는다)는 목록, 표, 화상, 초본문련결을 비롯한 기본HTML기능을 가지는 읽기전용의 QTextEdit파생클라스이다. Qt Assistant는 QTextBrowser에 의하여 사용자에게 문서를 표시한다.

QLineEdit QComboBox QSpinBox

QDateEdit QDateTimeEdit QTimeEdit

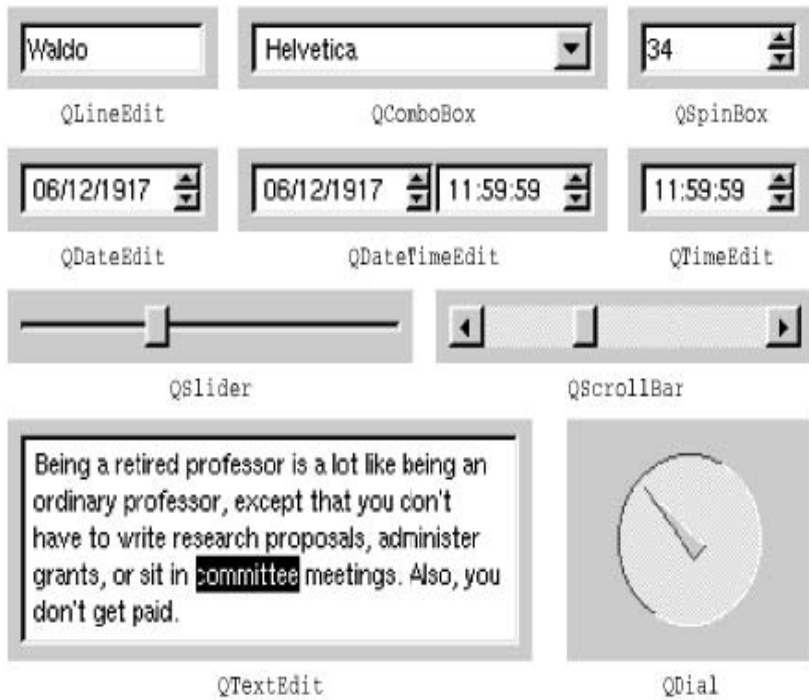
QSlider QScrollBar

QTextEdit QDial

그림 2-21. Qt의 입력창문부품들

Qt는 자료를 입력하는 창문부품들을 제공한다. QLineEdit는 입력마스크나 유효기(validator)에 의하여 그 입력을 제한할수 있다. QTextEdit는 대량적인 본문을 편집할수 있는 QScrollView의 파생클라스이다.

QColorDialog QFontDialog

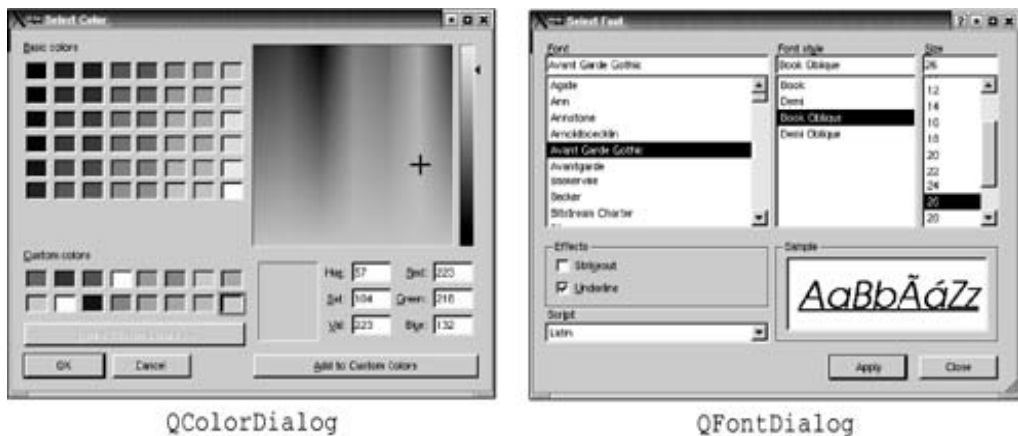



그림 2-22. Qt의 색대화칸와 서체대화칸

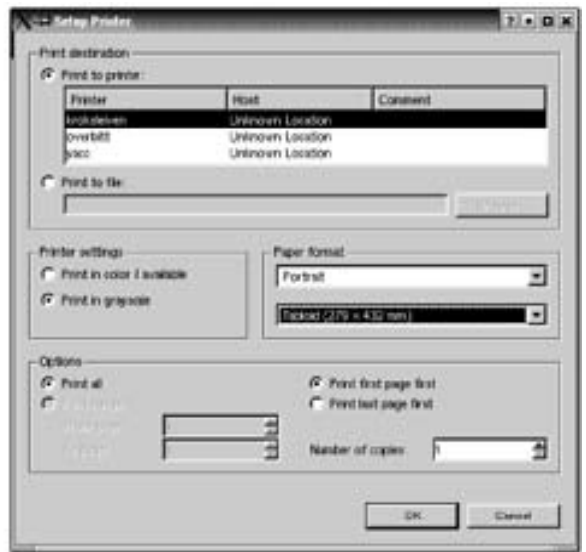
Qt는 사용자가 색, 서체 혹은 파일을 간단히 선택하거나 문서를 인쇄하게 하는 공통대화칸들의 표준묶음을 제공한다.

QFileDialog QPrintDialog

그림 2-23. Qt의 파일대화칸와 인쇄대화칸

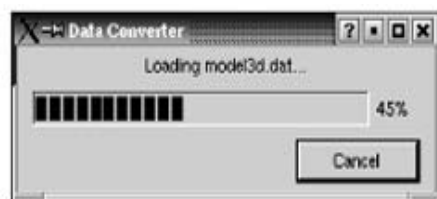
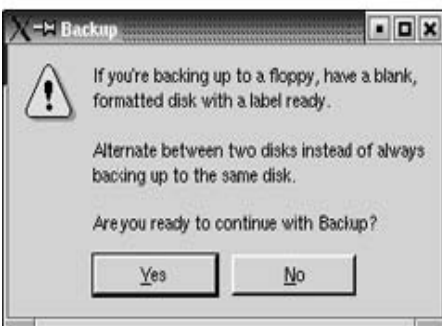
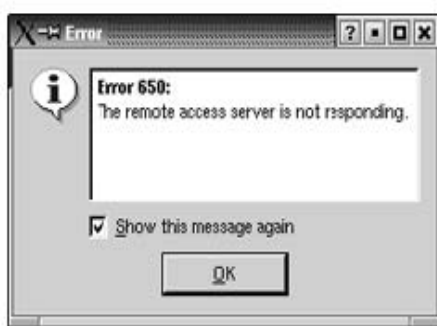
Windows와 Mac OS X에서 Qt는 될수록 자체의 공통대화칸이 아니라 원시대화칸을 사용한다.

QInputDialog QProgressDialog

QMessageBox QErrorMessage

그림 2-24. Qt의 귀환대화칸들

Qt는 표시한 통보문을 기억하는 만능통보창과 오유대화칸을 제공한다. 시간을 소비하는 조작의 진척상황은 QProgressDialog 혹은 초기에 보여준 QProgressBar에 의해 표시될수 있다. QInputDialog는 사용자가 한행의 본문이나 하나의 수를 요구할 때 아주 편리하다.

끝으로 QWizard는 위자드를 창조하는 틀거리를 제공한다. Qt Designer는 시각적으로 위자드를 창조하기 위한 Wizard형판을 제공한다.

Form1

System Requirements

Before starting this wizard, be sure the following steps have been completed:

- Installation of the Operating System
- Installation of the Application Server
- Filling out the Configuration Worksheets

If any of the above steps still need to be completed, click **Cancel** to close this wizard and restart it when you have completed all the requirements.

If all the requirements have been met, click **Next** to continue.

Cancel | < Back | Next >

그림 2-25. Qt의 QWizard대화칸

사용할 준비가 되여있는 수많은 기능이 기본창문부품과 공통대화칸들에 의해 제공된다. 더 전문화된 요구들은 흔히 신호들을 처리부들에 련결하고 처리부들에서 사용자정의동작을 실현함으로써 충족시킨다.

일부 경우에서 사용자정의창문부품을 창조해야 한다. Qt는 이것을 간단히 수행하며 사용자정의창문부품들은 Qt의 기본창문부품들처럼 같은 가동환경에 의존하지 않는 그리기기능을 모두 호출할수 있다. 지어는 사용자정의창문부품들을 Qt Designer에 통합하여 Qt의 기본창문부품들과 같은 방법으로 사용할수 있다. 제5장에서는 사용자정의창문부품창조방법을 설명한다.

제3장. 기본창문의 만들기

이 장에서는 Qt를 리용하여 기본창문을 창조하는 방법을 설명한다. 마감에는 응용프로그람의 완전한 사용자대면부를 만들고 차림표, 도구띠, 상태띠, 응용프로그람이 요구하는 대화칸들을 완성할수 있다.

응용프로그람의 기본창문은 응용프로그람의 사용자대면부를 건설하는 틀거리를 제공한다. 그림 3-1에 보여준 표계산프로그람의 기본창문은 이 장의 기초이다. 표계산프로그람은 제2장에서 창조한 Find, Go-to-Cell, Sort대화칸들을 사용한다.

대부분의 GUI응용프로그람들의 리면에는 기본기능을 제공하는 코드본체 즉 례를 들면 파일을 읽고쓰고 사용자대면부에 표시된 자료를 처리하는 코드본체가 놓여있다. 4장에서는 표계산프로그람을 다시 리용하여 그러한 기능을 실현하는 방법을 설명한다.

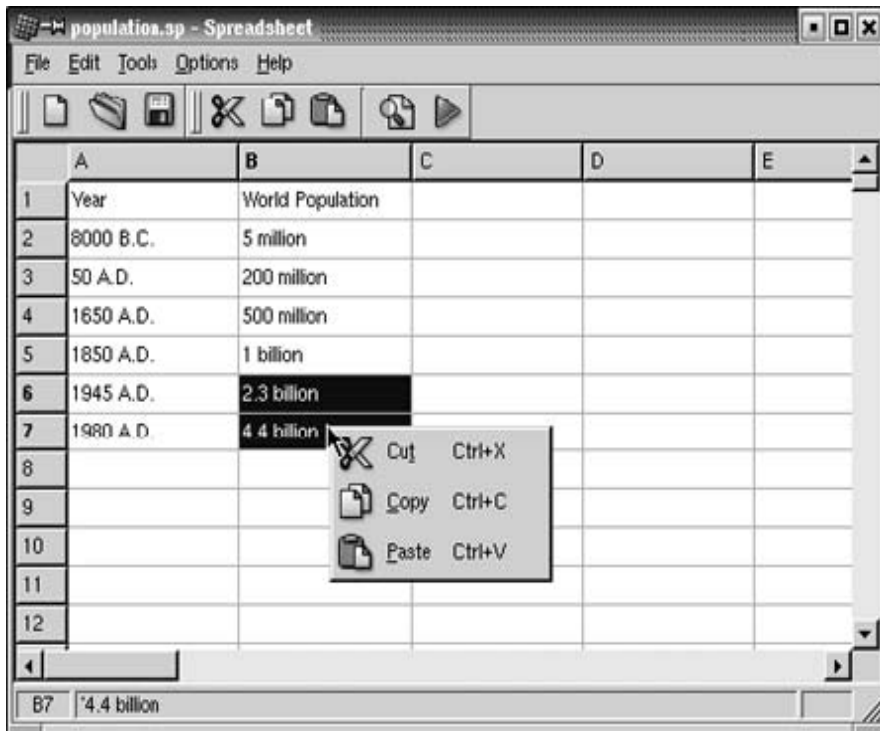



그림 3-1. 표계산프로그람

제1절. QMainWindow의 파생클라스만들기

응용프로그람의 기본창문은 QMainWindow의 파생클라스로서 창조한다. 제2장에서 본 대화칸을 창조하는 대부분의 기술은 QDialog와 QMainWindow가 둘다 QWidget로부터 계승되므로 기본창문의 창조와도 관련되여있다.

기본창문은 Qt Designer로 창조할수 있으나 이 장에서는 코드로 실현하는 방법을 보여준다.

표계산프로그람에서 기본창문의 원천코드는 mainwindow.h와 mainwindow.cpp에 들어있다. 머리부파일은 다음과 같다.

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H
#include <qmainwindow.h>
#include <qstringlist.h>
class QAction;
class QLabel;
class FindDialog;
class Spreadsheet;
class MainWindow : public QMainWindow
{
    Q_OBJECT
public:
    MainWindow(QWidget *parent = 0, const char *name = 0);
protected:
    void closeEvent(QCloseEvent *event);
    void contextMenuEvent(QContextMenuEvent *event);
```

MainWindow클라스를 QMainWindow의 파생클라스로서 정의한다. 이것은 자체의 신호와 처리부를 제공하므로 Q_OBJECT마크로를 포함한다.

closeEvent()함수는 사용자가 창문을 닫을 때 자동적으로 호출되는 QWidget의 가상함수이다. 이것은 MainWindow에서 사용자에게 표준질문 "Do you want to save your changes?"를 표시하고 디스크에 사용자의 선택을 보관하도록 재정의한다.

마찬가지로 contextMenuEvent()함수는 사용자가 창문부품을 오른단추로 찰칵하거나 가동환경에 고유한 차림표건을 누를 때 호출된다. 이것은 MainWindow에서 상황차림표를 펼치도록 재정의된다.

```
private slots:
    void newFile();
    void open();
    bool save();
    bool saveAs();
    void find();
    void goToCell();
    void sort ();
    void about();
```

File|New와 Help|About와 같은 일부 차림표선택들은 MainWindow에서 비공개처리부들로서 실현된다. 대부분의 처리부들은 돌림값으로서 void를 가지지만 save()와 saveAs()는 bool을 돌려준다. 돌림값은 신호에 응답하여 처리부가 실행될 때 무시되지만 함수로서 처리부를 호출할 때 돌림값은 보통의 C++함수를 호출할 때처럼 사용할수 있다.

```
    void updateCellIndicators();
    void spreadsheetModified();
    void openRecentFile(int param);
private:
    void createActions();
    void createMenus();
    void createToolBars();
    void createStatusBar();
    void readSettings();
    void writeSettings();
    bool maybeSave();
    void loadFile(const QString &fileName);
    void saveFile(const QString &fileName);
    void setCurrentFile(const QString &fileName);
    void updateRecentFileItems();
    QString strippedName(const QString &fullFileName);
```

기본창문은 사용자대면부를 유지하기 위한 많은 비공개처리부들과 여러개의 비공개함수들을 요구한다.

```
    Spreadsheet *spreadsheet;
    FindDialog *findDialog;
    QLabel *locationLabel;
    QLabel *formulaLabel;
    QLabel *modLabel;
    QStringList recentFiles;
    QString curFile;
    QString fileFilters;
    bool modified;
    enum { MaxRecentFiles = 5 };
    int recentFileIds[MaxRecentFiles];
    QPopupMenu *fileMenu;
    QPopupMenu *editMenu;
```

```
    QPopupMenu  *selectSubMenu;
    QPopupMenu  *toolsMenu;
    QPopupMenu  *optionsMenu;
    QPopupMenu  *helpMenu;
    QToolBar  *fileToolBar;
    QToolBar  *editToolBar;
    QAction  *newAct;
    QAction  *openAct;
    QAction  *saveAct;
    ...
    QAction *aboutAct;
    QAction *aboutQtAct;
};
#endif
```

또한 MainWindow는 비공개처리부, 비공개함수들과 함께 수많은 비공개변수들을 가진다. 이것들은 사용할 때 설명한다.

이제 그 실현을 고찰하자.

```
#include  <qaction.h>
#include  <qapplication.h>
#include  <qcombobox.h>
#include  <qfiledialog.h>
#include  <qlabel.h>
#include  <qlineedit.h>
#include  <qmenubar.h>
#include  <qmessagebox.h>
#include  <qpopupmenu.h>
#include  <qsettings.h>
#include  <qstatusbar.h>
#include  "cell.h"
#include  "finddialog.h"
#include  "gotocelldialog.h"
#include  "mainwindow.h"
#include  "sortdialog.h"
#include "spreadsheet.h"
```

파생클라스에서 사용하는 Qt클라스들의 머리부파일들과 일부 사용자정의머리부파일 특히

2장의 finddialog.h, gotocelldialog.h, sortdialog.h들을 포함한다.

```
MainWindow::MainWindow(QWidget *parent, const char *name) : QMainWindow(parent, name)
{
    spreadsheet = new Spreadsheet(this);
    setCentralWidget(spreadsheet);
    createActions();
    createMenus();
    createToolBars();
    createStatusBar();
    readSettings();
    setCaption(tr("Spreadsheet"));
    setIcon(QPixmap::fromMimeSource("icon.png"));
    findDialog = 0;
    fileFilters = tr("Spreadsheet files (*.sp)");
    modified = false;
}
```

구성자에서는 Spreadsheet창문부품을 창조하여 기본창문의 중심창문부품으로 설정하는것으로 시작한다. 중심창문부품은 도구띠들과 상태띠사이의 구역을 차지한다. Spreadsheet클라스는 표계산식들의 유지와 같은 표계산능력을 갖춘 QTable의 파생클라스이다. 그것을 4장에서 실현한다.

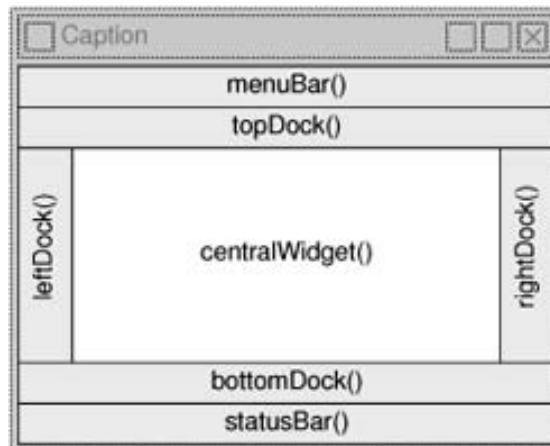



그림 3-2. QMainWindow의 구성창문부품들

그다음 비공개함수들인 createActions(), createMenus(), createToolBars(), createStatusBar()를 호출하여 기본창문의 나머지를 창조한다. 또한 비공개함수readSettings()를 호출하여 기억되여있는 응용프로그람의 환경설정을 읽어들인다.

창문의 그림기호를 PNG파일인 icon.png로 설정한다. Qt는 BMP, GIF, JPEG, MNG, PNG, PNM,

XBM 및 XPM을 비롯한 수많은 화상형식을 제공한다. QWidget::setIcon()을 호출하여 창문의 윈쪽웃구석에 그림기호가 표시되게 설정한다. 가동환경에 의존하지 않고 탁상우에 나타나는 응용프로그람그림기호를 설정하는 방법은 없다.

일반적으로 GUI응용프로그람들은 수많은 화상들을 사용하고 일부 화상은 여러개의 다른 상황에서 사용된다. Qt는 응용프로그람에 화상을 제공하는 여러가지 방법을 가지고있다.

가장 일반적인 방법은 다음과 같다.

· 파일들에 화상을 보관하였다가 실행시에 적재하기.

· 원천코드에 XPM파일들을 포함하기. (이것은 XPM파일들도 유효한 C++파일들이므로 제대로 동작한다.)

· Qt의 《화상집합》기구의 사용.

여기서는 실행시에 파일을 적재하는것보다 간단하면서 효과적인 화상집합수법을 리용한다. 이것은 유지되여있는 파일형식으로 작업한다. 화상들은 images라고 부르는 보조등록부안의 원천나무에 보관된다. 응용프로그람의 .pro파일에 다음의 코드

```
IMAGES = images/icon.png \
    images/new.png \
    images/open.png \
    ...
    images/find.png \
    images/gotocell.png
```

를 추가함으로써 uic가 지정된 화상모두에 대한 자료를 포함하는 C++원천코드파일을 생성하게 한다. 그다음 자료는 응용프로그람의 실행파일로 콤파일되고 QPixmap::fromMimeSource()에 의하여 얻을수 있다. 이것은 그림기호와 기타 화상들이 적재되여 항상 실행가능상태에 있다는 우점이 있다.

Qt Designer에 의하여 대화칸은 물론 기본창문을 창조한다면 그것을 리용하여 .pro파일을 조종할수 있고 화상들을 화상집합에 시각적으로 추가할수 있다.

제2절. 차림표와 도구띠의 만들기

대부분의 GUI응용프로그람들은 차림표와 도구띠를 모두 제공하며 일반적으로 차림표와 도구띠는 같은 지령들을 포함한다. 차림표는 사용자들이 응용프로그람을 조사하고 새로운 기능을 호출할수 있게 하고 도구띠는 흔히 사용하는 기능에 대한 고속호출을 제공한다.

Qt는 《작용》개념을 통하여 차림표와 도구띠의 프로그람작성을 단순화한다. 작용(action)은 하나의 차림표, 하나의 도구띠, 혹은 차림표와 도구띠에 모두 추가할수 있는 항목이다. Qt에서 차림표와 도구띠는 다음과 같은 단계를 통하여 창조된다.

- 작용들을 창조한다.
- 작용들을 차림표에 추가한다.
- 작용들을 도구띠에 추가한다.

표계산프로그람에서 작용들은 createActions()에서 창조된다.

```
void MainWindow::createActions()
{
    newAct = new QAction(tr("&New"), tr("Ctrl+N"), this);
    newAct->setIconSet(QPixmap::fromMimeSource("new.png"));
    newAct->setStatusTip(tr("Create a new spreadsheet file"));
    connect(newAct, SIGNAL(activated()), this, SLOT(newFile()));
```

New작용은 작용이름(New), 지름건(Ctrl+N), 부모(기본창문), 그림기호(new.png), 그리고 상태암시를 가진다. 작용의 activated()신호를 기본창문의 비공개성원 newFile()처리부에 련결한다. 이것은 다음 절에서 실현한다. 련결이 없으면 사용자가 File|New차림표항목을 선택하거나 *New* 도구띠단추를 찰칵할 때 아무런 반응도 없다.

File, Edit, Tools차림표의 다른 작용들은 New작용과 아주 비슷하다.

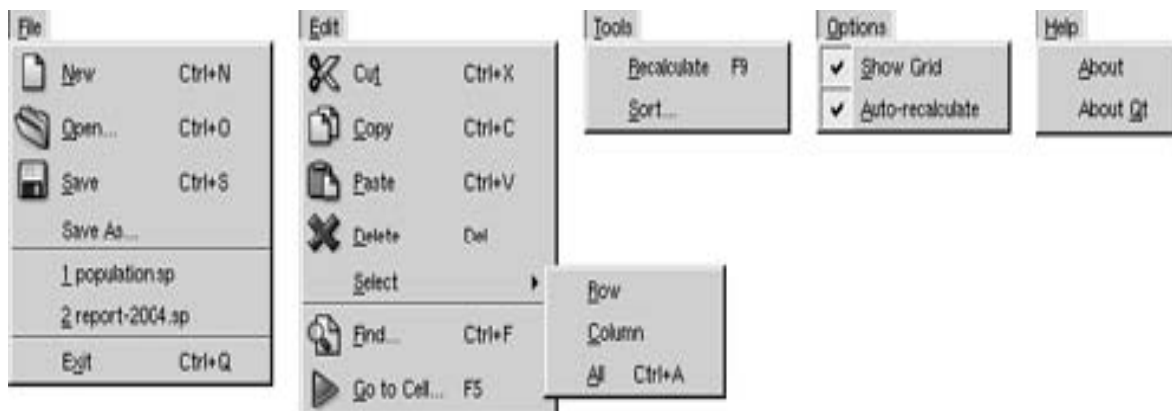



그림 3-3. 표계산프로그람의 차림표

Options차림표의 Show Grid작용은 다르다.

```
showGridAct = new QAction(tr("&Show Grid"), 0, this);
showGridAct->setToggleAction(true);
showGridAct->setOn(spreadsheet->showGrid());
showGridAct->setStatusTip(tr("Show or hide the spreadsheet's grid"));
    connect(showGridAct, SIGNAL(toggled(bool)),  spreadsheet, SLOT(setShowGrid()));
```

Show Grid는 절환(toggle)작용이다. 이것은 차림표에서 검사표식으로 표시되고 도구띠에서 절환단추로서 실현된다. 작용이 on일 때 Spreadsheet부분품은 살창을 현시한다. 작용을 Spreadsheet부분품의 기정값으로 초기화하여 그것들을 기동시에 동기시킨다. 그다음 Show Grid 작용의 toggled(bool)신호를 QTable로부터 계승하는 Spreadsheet부분품의 setShowGrid(bool)처리부에 련결한다. 일단 이 작용이 차림표 혹은 도구띠에 추가되면 사용자는 살창의 on/off를 절환할수 있다.

Show Grid와 Auto-recalculate작용들은 독립적인 절환작용들이다. 또한 QAction는

QActionGroup파생클라스를 통하여 호상 배타적인 작용들을 제공한다.

```
 aboutQtAct = new QAction(tr("About &Qt"), 0, this);
 aboutQtAct->setStatusTip(tr("Show the Qt library의 About box"));
 connect(aboutQtAct, SIGNAL(activated()), qApp, SLOT(aboutQt()));
}
```

About Qt에서는 qApp대역변수를 통하여 호출할수 있는 QApplication객체의 aboutQt()처리부를 리용한다.

이제는 작용들을 창조하였으므로 작용들을 호출할수 있는 차림표체계의 작성에로 넘어갈수 있다.

About Qt

About Qt

This program uses Qt version 3.2.0.

Qt is a C++ toolkit for multiplatform GUI & application development.

Qt provides single-source portability across MS Windows, Mac OS X, Linux, and all major commercial Unix variants.
Qt is also available for embedded devices.

Qt is a Trolltech product. See http://www.trolltech.com/qt/ for more information.

OK

그림 3-4. About Qt

```
void MainWindow::createMenus()
{
    fileMenu = new QPopupMenu(this);
    newAct->addTo(fileMenu);
    openAct->addTo(fileMenu);
    saveAct->addTo(fileMenu);
    saveAsAct->addTo(fileMenu);
    fileMenu->insertSeparator();
    exitAct->addTo(fileMenu);
   for (int i = 0; i < MaxRecentFiles; ++i)
        recentFileIds[i] = -1;
```

Qt에서 모든 차림표는 QPopupMenu의 실례들이다. File차림표를 창조한 다음 거기에 New, Open, Save, Save As, Exit작용들을 추가한다. 분리선을 추가하여 밀접히 련관된 항목들을 모두 시각적으로 묶는다. for순환을 리용하여 recentFilesIds배렬을 초기화한다. 다음 절에서 File차림

표처리부들을 실현할 때 recentFilesIds을 리용한다.

```
editMenu = new QPopupMenu(this);
cutAct->addTo(editMenu);
copyAct->addTo(editMenu);
pasteAct->addTo(editMenu);
deleteAct->addTo(editMenu);
selectSubMenu = new QPopupMenu(this);
selectRowAct->addTo(selectSubMenu);
selectColumnAct->addTo(selectSubMenu);
selectAllAct->addTo(selectSubMenu);
editMenu->insertItem(tr("&Select"), selectSubMenu);
editMenu->insertSeparator();
findAct->addTo(editMenu);
goToCellAct->addTo(editMenu);
```

Edit차림표는 하나의 보조차림표를 포함한다. 보조차림표는 그것이 속하는 차림표처럼 QPopupMenu이다. 단순히 this를 부모로 가지는 보조차림표를 창조하여 그것을 표시하려는 Edit차림표에 삽입한다.

```
    toolsMenu = new QPopupMenu(this);
    recalculateAct->addTo(toolsMenu);
    sortAct->addTo(toolsMenu);
    optionsMenu = new QPopupMenu(this);
    showGridAct->addTo(optionsMenu);
    autoRecalcAct->addTo(optionsMenu);
    helpMenu = new QPopupMenu(this);
    aboutAct->addTo(helpMenu);
    aboutQtAct->addTo(helpMenu);
    menuBar()->insertItem(tr("&File"), fileMenu);
    menuBar()->insertItem(tr("&Edit"), editMenu);
    menuBar()->insertItem(tr("&Tools"), toolsMenu);
    menuBar()->insertItem(tr("&Options"), optionsMenu);
    menuBar()->insertSeparator();
    menuBar()->insertItem(tr("&Help"), helpMenu);
}
```

류사한 방법으로 Tools, Options, Help차림표들을 만들어 모든 차림표를 차림표띠에 삽입한다. QMainWindow::menuBar()함수는 QMenuBar의 지적자를 돌려준다. (차림표띠는 menuBar()가

처음으로 호출될 때 창조된다.) Options와 Help차림표사이에 분리선을 삽입한다. Motif형식에서 분리선은 Help차림표를 오른쪽에 배치하며 다른 형식들에서 분리선은 무시된다.

그림 3-5. Motif와 Windows형식들에서 차림표띠

도구띠의 창조는 차림표창조와 아주 비슷하다.

```
void MainWindow::createToolBars()
{
    fileToolBar = new QToolBar(tr("File"), this);
    newAct->addTo(fileToolBar);
    openAct->addTo(fileToolBar);
    saveAct->addTo(fileToolBar);
    editToolBar = new QToolBar(tr("Edit"), this);
    cutAct->addTo(editToolBar);
    copyAct->addTo(editToolBar);
    pasteAct->addTo(editToolBar);
    editToolBar->addSeparator();
    findAct->addTo(editToolBar);
    goToCellAct->addTo(editToolBar);
}
```

File도구띠와 Edit도구띠를 창조한다. 튀여나오기차림표처럼 도구띠는 분리선을 가질수 있다.

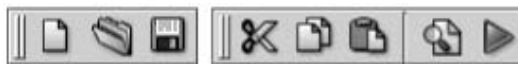



그림 3-6. 표계산프로그람의 도구띠

이제는 차림표와 도구띠를 만들었으므로 상황차림표를 추가하여 대면부를 완성한다.

```
void MainWindow::contextMenuEvent(QContextMenuEvent *event)
{
    QPopupMenu contextMenu(this);
    cutAct->addTo(&contextMenu);
    copyAct->addTo(&contextMenu);
    pasteAct->addTo(&contextMenu);
    contextMenu.exec(event->globalPos());
```

}

사용자가 오른쪽 마우스단추를 찰칵할 때(혹은 일부 건반우에 있는 Menu건을 누를 때), 《상황차림표》사건이 창문부품에 전송된다. QWidget::contextMenuEvent()함수를 재정의하여 이 사건에 응답할수 있으며 현재의 마우스유표위치에 상황차림표를 펼친다.

그림 3-7. 표계산프로그람의 상황차림표

신호와 처리부처럼 사건은 Qt프로그람작성의 기초개념이다. 사건(event)은 Qt의 핵심에 의해 생성되여 마우스찰칵, 건누르기, 크기조절요구 및 류사한 경우를 통보한다. 여기서 수행한 것처럼 가상함수들을 재정의하여 사건들을 조종할수 있다.

MainWindow에 차림표의 작용들을 보관하였으므로 여기서 상황문맥차림표를 실현하기로 하였지만 Spreadsheet에 실현할수도 있다. 사용자가 Spreadsheet창문부품을 오른쪽 단추로 찰칵할 때 Qt는 상황차림표사건을 그 창문부품에 우선 전송한다. Spreadsheet가 contextMenuEvent()를 재정의하여 사건을 조종하면 사건은 거기서 정지하며 그렇지 않으면 부모(MainWindow)에 송신된다. 사건은 7장에서 완전히 설명된다.

상황차림표사건처리함수는 탄창에 변수로서 창문부품(QPopupMenu)을 창조하므로 지금까지 보아온 모든 코드와는 다르다. new와 delete를 사용하여 간단히 창조할수 있다.

```
QPopupMenu *contextMenu = new QPopupMenu(this);
cutAct->addTo(contextMenu);
copyAct->addTo(contextMenu);
pasteAct->addTo(contextMenu);
contextMenu->exec(event->globalPos());
delete contextMenu;
```

또 하나의 중요한 코드는 exec()호출이다. QPopupMenu::exec()는 주어진 화면위치에 튀여나오기차림표를 표시하고 사용자가 차림표항목을 선택할 때까지(혹은 튀여나오기차림표가 없어질 때까지) 기다린다. 이 시점에서 QPopupMenu객체는 자기의 목적을 달성하였으므로 그것을 해체할수 있다. QPopupMenu객체가 탄창에 배치되면 그것은 함수의 끝에서 자동적으로 해체되고 그렇지 않으면 delete를 호출해야 한다.

이제는 차림표와 도구띠의 사용자대면부를 완성하였다. 아직도 모든 처리부를 실현하지 못했고 File차림표의 최근에 연 파일들을 조종하는 코드를 쓰지 못했다. 다음의 두개 절에서 그 부분을 완성한다.

제3절. File차림표의 실현

이 절에서는 File차림표가 동작하게 하는데 필요한 처리부와 비공개함수들을 실현한다.

```
void MainWindow::newFile()
{
    if (maybeSave()) {
        spreadsheet->clear();
        setCurrentFile("");
    }
}
```

newFile()처리부는 사용자가 File|New차림표를 찰칵하거나 New도구띠단추를 찰칵할 때 호출된다. maybeSave()비공개함수는 보관안된 변경이 있으면 사용자에게 "Do you want to save your changes?"라고 묻는다. 사용자가 Yes 혹은 No(문서를 보관할 때 Yes)를 선택하면 true를 돌려주고 사용자가 Cancel을 선택하면 false를 돌려준다. setCurrentFile()비공개함수는 새로 만든 파일은 제목이 없으므로 창문제목을 기정제목으로 설정한다.

```
bool MainWindow::maybeSave()
{
 if (modified) {
     int ret = QMessageBox::warning(this, tr("Spreadsheet"),
                       tr("The document has been modified.\n"
                          "Do you want to save your changes?"),
                       QMessageBox::Yes | QMessageBox::Default, QMessageBox::No,
                       QMessageBox::Cancel | QMessageBox::Escape);
     if (ret == QMessageBox::Yes)
          return save();
    else if (ret == QMessageBox::Cancel)
          return false;
 }
 return true;
}
```

maybeSave()에서는 그림 3-8에 보여주는 통보창을 현시한다. 통보창은 Yes, No, Cancel단추를 가진다. QMessageBox::Default는 Yes를 기정단추로 만든다. QMessageBox::Escape는 Esc건을 Cancel과 같은 의미로 만들어준다.

그림 3-8. 변경된 문서에 파일의 보관을 물어보는 통보창

warning()의 호출은 첫눈에 좀 복잡해보이지만 일반문법은 간단하다.

QMessageBox::warning(*parent*,*caption*,*messageText*, *button0*, *button1*, ...);

또한 QMessageBox는 warning()처럼 동작하지만 각이한 그림기호를 현시하는 information(), question(), critical()을 제공한다.

그림 3-9. 통보창그림기호

```
void MainWindow::open()
{
    if (maybeSave()) {
        QString fileName = QFileDialog::getOpenFileName(".", fileFilters, this);
    if (!fileName.isEmpty())
        loadFile(fileName);
    }
}
```

open()처리부는 File|Open에 대응된다. newFile()처럼 open()처리부는 우선 maybeSave()를 호출하여 보관하지 않은 변경을 조종한다. 그다음 정적편의함수 QFileDialog::getOpenFileName()에 의하여 파일이름을 얻는다. 이 함수는 파일대화칸을 펼치고 사용자가 파일을 선택하면 파일이름(혹은 사용자가 Cancel을 찰칵하면 빈문자렬)을 돌려준다.

getOpenFileName()함수에 3개의 인수를 준다. 첫째 인수는 기동하려는 등록부, 이 경우에 현재등록부를 함수에 알린다. 둘째 인수 fileFilters는 파일려과기들을 지정한다. 파일려과기는 파일형이름과 확장자로 이루어진다. MainWindow구성자에서 fileFilters는 다음과 같이 초기화된다.

```
fileFilters = tr("Spreadsheet files (*.sp)");
```

Spreadsheet의 원시파일형식과 함께 반점으로 구분된 값(CSV, comma-separated value)들의 파일, Lotus 1-2-3파일을 유지하려면 변수를 다음과 같이 초기화하여야 한다.

```
fileFilters = tr("Spreadsheet files (*.sp)\n"
```

```
"Comma-separated values files (*.csv)\n" "Lotus 1-2-3 files (*.wk?)");
```

끝으로 getOpenFileName()의 셋째 인수는 펼쳐지는 QFileDialog가 기본창문의 자식으로 되여야 한다는것을 지정한다.

부모-자식관계는 대화칸에서 다른 창문부품에서와 같은 의미가 아니다. 대화칸은 늘 제일 웃준위 창문부품(자체가 창문)이지만 그것이 부모를 가지면 기정으로 부모의 우에 중심에 배치된다. 또한 자식대화칸은 부모의 과제띠항목을 공유한다.

```
void MainWindow::loadFile(const QString &fileName)
{
    if (spreadsheet->readFile(fileName)) {
        setCurrentFile(fileName);
        statusBar()->message(tr("File loaded"), 2000);
    } else {
        statusBar()->message(tr("Loading canceled"), 2000);
    }
}
```

loadFile()비공개함수는 파일을 적재하기 위하여 open()에서 호출된다. 최근에 연 파일들을 적재하는데 같은 기능이 필요하므로 그것을 독립적인 함수로 만든다.

Spreadsheet::readFile()에 의하여 디스크로부터 파일을 읽어들인다. 적재에서 성공하면 setCurrentFile()을 호출하여 창문의 제목을 갱신한다. 그렇지 않으면 Spreadsheet::loadFile()은 사용자에게 통보창을 통하여 문제점에 대하여 미리 통지한다. 일반적으로 저준위부분품들이 오유에 대한 정확한 내용을 제공할수 있으므로 오유통보문을 내게 하는것이 좋다.

두 경우에 2000ms(2s)동안 상태띠에 통보문을 현시하여 응용프로그람이 수행하는 일을 사용자가 알수 있게 한다.

```
bool MainWindow::save()
{
    if (curFile.isEmpty()) {
        return saveAs();
    } else {
        saveFile(curFile);
        return true;
    }
}
void MainWindow::saveFile(const QString &fileName)
{
    if (spreadsheet->writeFile(fileName)) {
```

```
            setCurrentFile(fileName);
            statusBar()->message(tr("File saved"), 2000);
        } else {
            statusBar()->message(tr("Saving canceled"), 2000);
        }
    }
```

save()처리부는 File|Save에 대응된다. 파일이 이전에 열렸거나 이미 보관되였기때문에 파일에 이미 이름이 있으면 save()는 그 이름으로 saveFile()를 호출하고 그렇지 않으면 단순히 saveAs()를 호출한다.

```
    bool MainWindow::saveAs()
    {
        QString fileName = QFileDialog::getSaveFileName(".", fileFilters, this);
        if (fileName.isEmpty())
            return false;
        if (QFile::exists(fileName)) {
            int ret = QMessageBox::warning(this, tr("Spreadsheet"), tr("File %1 already exists. \n"
                                "Do you want to overwrite it?")
                            .arg(QDir::convertSeparators(fileName)),
                            QMessageBox::Yes | QMessageBox::Default,
                            QMessageBox::No | QMessageBox::Escape);
            if (ret == QMessageBox::No)
                return true;
        }
        if (!fileName.isEmpty())
            saveFile(fileName);
        return true;
    }
```

saveAs()처리부는 File|Save As에 대응된다. QFileDialog::getSaveFileName()를 호출하여 사용자로부터 파일이름을 얻는다. 사용자가 Cancel을 찰칵하면 false를 돌려주는데 이것은 maybeSave()까지 전달된다. 그렇지 않으면 돌아온 파일이름은 새로운 이름이거나 현존파일의 이름이다. 현존파일의 경우에는 QMessageBox::warning()를 호출하여 그림 3-10에 보여주는 통보창을 현시한다.

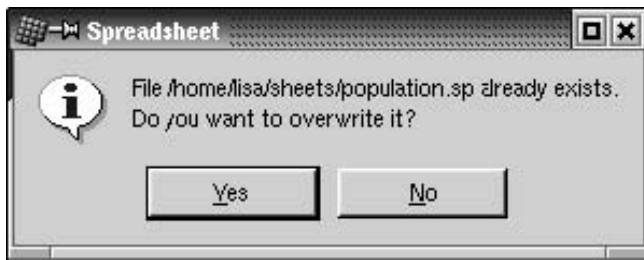



그림 3-10. 겹쳐쓰겠는가를 물어보는 통보창

통보창에 넘긴 본문은 다음과 같다.

```
tr("File %1 already exists\n"
     "Do you want to override it?")
     .arg(QDir::convertSeparators(fileName))
```

QString::arg()함수는 "%*n*"파라메터를 그 인수로 바꾸고 결과문자렬을 돌려준다. 례를 들면 파일이름이 A:\tab04.sp이면 우의 코드는 다음과 같다.

```
"File A:\\tab04.sp already exists.\n"
"Do you want to override it?"
```

이것은 응용프로그람이 다른 언어로 번역되지 않는것을 전제로 하고있다. QDir::convertSeparators()호출은 Qt가 이식가능한 등록부분리기호로 사용하는 사선들을 가동환경에 고유한 분리기호(Unix와 Mac OS X에서 '/', Windows에서 '\' on)로 변환한다.

```
void MainWindow::closeEvent(QCloseEvent *event)
{
    if (maybeSave()) {
        writeSettings();
        event->accept();
    } else {
        event->ignore();
    }
}
```

사용자가 File|Exit를 찰칵하거나 창문제목띠의 ×를 찰칵할 때 QWidget::close()처리부가 호출된다. 이것은 close사건을 창문부품에 송신한다. QWidget::closeEvent()를 재정의함으로써 기본창문을 닫으려는 시도를 받아들이고 창문을 닫으려고 하는가 아닌가를 결정할수 있다.

보관하지 않은 변경내용이 있는데 사용자가 Cancel을 선택하면 그 사건을 《무시》하고 창문은 그대로 남아있다. 그렇지 않으면 사건을 받아들이고 Qt에서는 창문을 닫으며 응용프로그람을 완료한다.

```
void MainWindow::setCurrentFile(const QString &fileName)
{
```

```
        curFile = fileName;
        modLabel->clear();
        modified = false;
       if (curFile.isEmpty()) {
           setCaption(tr("Spreadsheet"));
       } else {
           setCaption(tr("%1 -%2").arg(strippedName(curFile)).arg(tr("Spreadsheet")));
           recentFiles.remove(curFile);
           recentFiles.push_front(curFile);
           updateRecentFileItems();
       }
   }
   QString MainWindow::strippedName(const QString &fullFileName)
   {
       return QFileInfo(fullFileName).fileName();
   }
```

setCurrentFile()에서는 편집중에 있는 파일의 이름을 보관하는 curFile비공개변수를 설정하고 MOD상태지시자를 지우고 제목을 갱신한다. 두개의 %*n*파라메터에 대하여 arg()를 사용한다. arg()의 첫 호출은 "%1"로 교체되고 둘째 호출은 "%2"로 교체된다.

```
setCaption(strippedName(curFile) + tr(" -Spreadsheet"));
```

와 같이 쓰는것이 더 좋지만 arg()를 사용하면 번역기에 유연성을 준다. strippedName()에 의하여 파일의 경로를 삭제하고 파일이름을 사용자에게 편리하게 만든다.

파일이름이 있으면 응용프로그람의 최근에 연 파일목록인 recentFiles를 갱신한다. remove()를 호출하여 목록에서 파일이름을 삭제한 다음 push_front()를 호출하여 파일이름을 첫 항목으로서 추가한다. remove()의 호출은 무엇보다 먼저 중복을 피하는데 필요하다. 목록을 갱신한 후에 비공개함수 updateRecentFileItems()를 호출하여 File차림표전부를 갱신한다.

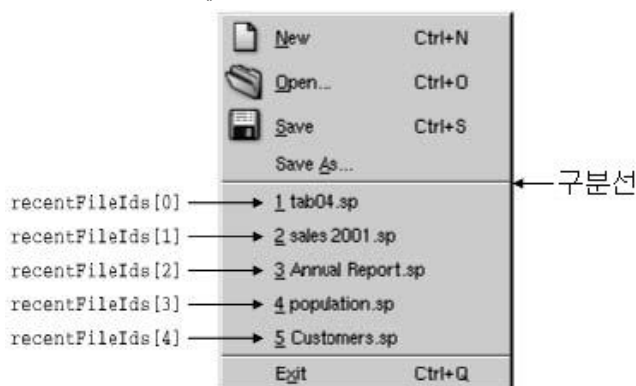



그림 3-11. 최근에 연 파일들이 있는 File차림표

centFiles변수는 QStringList형(QString의 목록)이다. 11장에서 QStringList와 같은 용기클라스들과 C++표준형판서고(STL)와의 관계를 자세히 설명한다.

이리하여 File차림표의 실현이 거의 완성되였다. 아직 실현하지 못한 한개의 함수와 한개의 처리부가 남아있다. 둘다 최근에 연 파일목록의 관리와 련관되여있다.

```
void MainWindow::updateRecentFileItems()
{
    while ((int)recentFiles.size() > MaxRecentFiles)
        recentFiles.pop_back();
    for (int i = 0; i < (int)recentFiles.size(); ++i) {
        QString text = tr("&%1 %2").arg(i + 1).arg(strippedName(recentFiles[i]));
        if (recentFileIds[i] == –1) {
            if (i == 0)
                fileMenu->insertSeparator(fileMenu->count() – 2);
            recentFileIds[i] = fileMenu->insertItem(text, this, SLOT(openRecentFile(int)),
                                                    0, –1, fileMenu->count() – 2);
            fileMenu->setItemParameter(recentFileIds[i], i);
        } else {
            fileMenu->changeItem(recentFileIds[i], text);
        }
    }
}
```

updateRecentFileItems()비공개함수는 최근에 연 파일들의 차림표항목을 갱신하기 위하여 호출된다. recentFiles목록에 허용된것보다 항목이 많지 않는가(MaxRecentFiles, mainwindow.h에서 5로 정의되였다.)를 확인하고 목록의 끝으로부터 남는 항목들을 삭제한다.

그다음 매개 항목들에 대하여 새로운 차림표항목을 창조하거나 혹은 존재하는 경우 현존항목을 재리용한다. 차림표항목을 처음으로 창조할 때 또한 분리선을 삽입한다. 이것을 createMenus()가 아니라 여기서 수행하여 한 행에 두개의 분리선을 절대로 현시하지 않도록한다. setItemParameter()호출을 간단히 설명한다.

이것은 updateRecentFileItems()에서 항목들을 창조하지만 절대로 항목들을 삭제하지 않는 것처럼 보일수 있다. 그것은 최근에 연 파일목록이 쎄숀기간에 절대로 줄어들지 않는다고 가정할수 있기때문이다.

QPopupMenu::insertItem()함수는 아래와 같은 형식을 가진다.

fileMenu->insertItem(*text, receiver, slot, accelerator, id, index*);

*text*는 차림표에 현시된 본문이다. strippedName()을 사용하여 파일이름으로부터 경로를 삭

제한다. 완전파일이름을 유지할수 있지만 그렇게 되면 File차림표가 너무 길어지게 된다. 완전한 파일차림표를 표시하려면 보조차림표에 최근에 연 파일들을 넣어야 한다.

*receiver*와 *slot*파라메터들은 사용자가 항목을 선택할 때 호출되여야 할 처리부를 지정한다. 실례에서는 MainWindow의 openRecentFile(int)처리부에 련결한다.

*accelerator*와 *id*에는 차림표항목에 지름건이 없으며 자동적으로 생성된 ID를 가진다는것을 의미하는 기정값들을 넘긴다. 생성된 ID를 recentFileIds배렬에 보관하여 후에 항목들을 호출할수 있다.

*index*는 항목을 삽입하려고 하는 위치이다. 값 fileMenu->count() –2를 넘김으로써 Exit항목의 분리선우에 그것을 삽입한다.

```
void MainWindow::openRecentFile(int param)
{
    if (maybeSave())
        loadFile(recentFiles[param]);
}
```

openRecentFile()처리부는 많은곳에서 호출되는 처리부이다. 이 처리부는 File차림표로부터 최근에 연 파일이 선택될 때 호출된다. int파라메터는 setItemParameter()로 초기에 설정한 값이다. recentFiles목록에 대한 첨수로서 그것들을 사용하는 방법으로 값들을 선택하였다.

안내항목들

ID	text	param
-32	1 tab04.sp	0
-33	2 sales 2001.sp	1
-34	3 Annual Report.sp	2
-35	4 population.sp	3
-36	5 Customers.sp	4

최근에 연 파일들

index	value
0	A:\tab04.sp
1	C:\sales 2001.sp
2	D:\Annual Report.sp
3	C:\population.sp
4	C:\Customers.sp

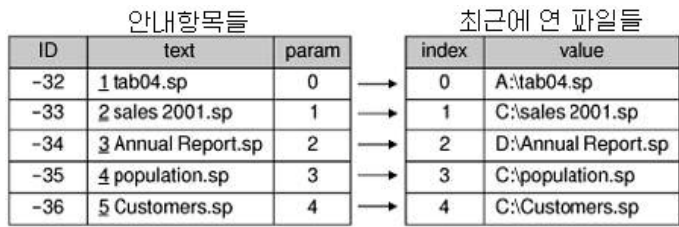



그림 3-12. 최근에 연 파일들의 관리

이것은 문제를 해결하는 하나의 방법이다. 이와 비슷한 다른 한가지 방법은 5개의 작용을 창조하여 5개의 개별적인 처리부들에 련결하는것이다.

제4절. 상태띠의 설정

차림표와 도구띠를 완성하였으므로 표계산프로그람의 상태띠를 작성할 준비가 되였다. 표준상태에서 상태띠는 3개의 지시자(indicator) 즉 현재세포의 위치, 현재세포의 식, MOD로 이루어진다. 또한 상태띠는 상태암시와 다른 림시통보문들을 현시하는데 쓰인다.

MainWindow구성자는 createStatusBar()를 호출하여 상태띠를 설정한다.

```
void MainWindow::createStatusBar()
{
     locationLabel = new QLabel(" W999 ", this);
    locationLabel->setAlignment(AlignHCenter);
    locationLabel->setMinimumSize(locationLabel->sizeHint());
```

```
    formulaLabel = new QLabel(this);
    modLabel = new QLabel(tr(" MOD "), this);
    modLabel->setAlignment(AlignHCenter);
    modLabel->setMinimumSize(modLabel->sizeHint());
    modLabel->clear();
    statusBar()->addWidget(locationLabel);
    statusBar()->addWidget(formulaLabel, 1);
    statusBar()->addWidget(modLabel);
    connect(spreadsheet, SIGNAL(currentChanged(int, int)), this, SLOT(updateCellIndicators()));
    connect(spreadsheet, SIGNAL(modified()), this, SLOT(spreadsheetModified()));
    updateCellIndicators();
}
```

QMainWindow::statusBar()함수는 상태띠의 지적자를 돌려준다.(상태띠는 statusBar()가 처음으로 호출될 때 창조된다.) 상태지시자는 필요할 때마다 본문이 변하는 QLabel이다. QLabel들을 구성할 때 this를 부모로서 넘기지만 QStatusBar::addWidget()가 그것들의 부모자식관계를 자동적으로 재설정하여 자식들을 만드므로 실제로 문제는 없다.

그림 3-13은 3개의 표식자가 각이한 공간요구를 가진다는것을 보여준다. 세포위치와 MOD지시자들은 아주 작은 공간을 요구하고 창문의 크기가 조절될 때 여분의 공간은 중간의 세포식(formula)이 차지하도록 한다. 이것은 QStatusBar::addWidget()호출에서 너비결수 1을 지정하여 달성된다. 다른 두개의 지시자들은 기정늘임결수 0을 가지는데 이것은 그것들의 크기가 변하지 않는다것을 의미한다.

QStatusBar가 지시자창문부품들을 배치할 때 QWidget::sizeHint()에 의해 주어진 매개 창문부품의 리상적인 크기를 고려하여 사용가능한 공간을 채우도록 창문부품들을 늘인다. 창문부품의 리상크기는 창문부품의 내용에 따라 달라진다. 위치와 MOD지시자들의 크기가 끊임없이 변하는것을 피하기 위하여 그것들의 최소크기를 매개 지시자에서 가능한 최대본문을 다 포함하는 폭("W999"와 "MOD")으로 설정한다. 또한 그것들의 배치를 AlignHCenter로 설정하여 본문이 수평으로 중심에 놓이도록 한다.

B4 =A1+A2+A3 MOD

표준

Open an existing spreadsheet file

상태암시

File saved

일시통보문

그림 3-13. 표계산프로그람의 상태띠

함수의 끝부근에서 Spreadsheet의 2개 신호를 MainWindow의 처리부 updateCellIndicators()와 spreadsheetModified()에 련결한다.

```
void MainWindow::updateCellIndicators()
{
    locationLabel->setText(spreadsheet->currentLocation());
    formulaLabel->setText(" " + spreadsheet->currentFormula());
}
```

updateCellIndicator()처리부는 세포위치와 세포식 지시자들을 갱신한다. 이것은 사용자가 세포유표를 새로운 세포로 이행할 때마다 호출된다. 또한 그 처리부는 createStatusBar()의 끝에서 보통의 함수로 사용되여 지시자들을 초기화한다. 이것은 Spreadsheet가 기동시에 currentChanged()신호를 발생하지 않으므로 필요하다.

```
void MainWindow::spreadsheetModified()
{
    modLabel->setText(tr("MOD"));
    modified = true;
    updateCellIndicators();
}
```

spreadsheetModified()처리부는 3개의 지시자를 모두 갱신하여 사건의 현재상태를 반영하고 변경된 변수를 true로 설정한다. (File차림표를 실현할 때 modified변수를 사용하여 보관하지 않은 변경이 있는가 결정하였다.)

제5절. 대화칸의 리용

이 절에서는 Qt에서 대화칸을 사용하는 방법 즉 대화칸을 창조하고 초기화하는 방법, 그 실행방법, 사용자의 선택에 응답하는 방법을 설명한다. 2장에서 창조한 Find, Go-to-Cell, Sort대화칸들을 사용할수 있게 한다. 또한 단순한 About칸도 창조한다.

우선 Find대화칸부터 시작한다. 사용자가 Spreadsheet기본창문과 Find대화칸을 마음대로 절환할수 있게 하려고 하므로 Find대화칸은 이행허용(modeless)대화칸이여야 한다. 이행허용창문은 응용프로그람의 다른 창문들과는 독립적으로 실행되는 창문이다.

이행허용대화칸이 창조될 때 보통 사용자의 교제에 응답하는 처리부들에 련결된 자기의 신호들을 가진다.

```
void MainWindow::find()
{
   if (!findDialog) {
        findDialog = new FindDialog(this);
        connect(findDialog, SIGNAL(findNext(const QString &, bool)),
             spreadsheet, SLOT(findNext(const QString &, bool)));
        connect(findDialog, SIGNAL(findPrev(const QString &, bool)),
             spreadsheet, SLOT(findPrev(const QString &, bool)));
```

```
    }
    findDialog->show();
    findDialog->raise();
    findDialog->setActiveWindow();
}
```

Find대화칸은 표계산프로그람에서 사용자가 본문을 탐색할수 있게 하는 창문이다. find()처리부는 사용자가 Edit|Find를 찰칵하여 Find대화칸을 펼칠 때 호출된다. 그 시점에서 여러가지 경우가 있다.

- 사용자가 처음으로 Find대화칸을 펼친 경우
- Find대화칸을 그전에 펼쳤는데 사용자가 닫은 경우
- Find대화칸을 그전에 펼쳤는데 아직 표시되여있는 경우.

Find대화칸이 이미 존재하지 않으면 그것을 창조하고 그의 findNext()와 findPrev()신호를 Spreadsheet의 대응하는 처리부들에 련결한다. 대화칸을 창조하지 않았다가 필요한 경우에 창조하면 프로그람의 기동을 빠르게 한다. 또한 대화칸을 한번이라도 사용하지 않으면 절대로 창조되지 않으며 이것은 시간과 기억기를 절약한다.

그다음 show(), raise(), setActiveWindow()를 호출하여 창문을 다른것들의 우에 표시하고 능동창문이 되도록 한다. show() 하나의 호출은 숨겨진 창문을 표시하는데 충분하며 Find대화칸의 창문이 이미 표시되여있을 때 호출되는 경우에 show()는 아무 일도 하지 않는다. 대화칸의 창문을 이전의 상태와는 관계없이 꼭대기에서 볼수 있게, 능동으로 만들어야 하므로 raise()와 setActiveWindow()를 호출해야 한다. 다른 방법은

```
if (findDialog->isHidden()) {
    findDialog->show();
} else {
    findDialog->raise();
    findDialog->setActiveWindow();
}
```

라고 쓰는것이다.

다음으로 Go-to-Cell대화칸을 고찰하자. 사용자가 대화칸을 펼치고 그것을 리용하며 Go-to-Cell대화칸으로부터 응용프로그람의 다른 창문으로 절환하지 않고 닫으려고 한다. 이것은 Go-to-Cell대화칸이 이행금지창문이라는것을 의미한다. 이행금지(modal)창문은 기동할 때 창문이 펼쳐진 다음부터 닫길 때까지 응용프로그람의 다른 처리나 교제를 금지하는 창문이다. Find대화칸을 제외하고 지금까지 사용한 모든 대화칸은 이행금지대화칸이다.

대화칸이 show()에 의해 펼쳐지면(미리 setModal()를 호출하여 이행금지로 만들지 않으면) 이행허용이고 exec()에 의하여 펼쳐지면 이행금지이다. 이행금지대화칸을 exec()에 의해 기동할 때 일반적으로 신호–처리부련결을 설정할 필요는 없다.

```
void MainWindow::goToCell()
{
    GoToCellDialog dialog(this);
    if (dialog.exec()) {
        QString str = dialog.lineEdit->text();
        spreadsheet->setCurrentCell(str.mid(1).toInt() -1, str[0].upper().unicode() -'A');
    }
}
```

QDialog::exec()함수는 대화칸을 받아들이면 true, 그렇지 않으면 false를 돌려준다. (2장에서 Qt Designer로 Go-to-Cell대화칸을 창조할 때 OK를 accept()에, Cancel을 reject()에 련결한것을 상기하시오.) 사용자가 OK를 찰칵하면 현재의 세포를 행편집기의 값으로 설정하고 사용자가 Cancel을 찰칵하면 exec()는 false를 돌려주고 아무 일도 하지 않는다.

QTable::setCurrentCell()함수는 2개의 인수 즉 행번호와 렬번호를 요구한다. 표계산프로그람에서 세포 A1은 cell (0, 0)이고 세포 B27은 cell (26, 1)이다. QLabel::text()가 돌려준 QString으로부터 행번호를 얻기 위해서는 QString::mid()(이것은 문자렬의 시작위치로부터 끝위치까지의 부분문자렬을 돌려준다.)에 의해 행번호를 얻고 QString::toInt()에 의하여 그것을 int로 변환하고 1을 덜어서 0으로부터 시작하는 셈의 행번호를 계산한다. 렬번호에 대해서는 문자렬의 대문자화된 첫 문자의 수값에서 'A'의 수값을 던다.

Find와 달리 Go-to-Cell대화칸은 탄창에 창조된다. 이행금지대화칸을 사용한 후에는 대화칸이 요구되지 않으므로 이것은 상황차림표에서처럼 이행금지대화칸의 일반적인 프로그람작성본보기이다.

다음으로 Sort대화칸을 고찰한다. Sort대화칸은 현재 선택된 구역을 지정된 렬들에 따라 사용자가 정렬하게 하는 이행금지대화칸이다. 그림 3-14는 1차정렬마당으로서 렬B, 2차정렬마당으로서 렬A를 가지는 정렬의 실례이다. (모두 자모순이다.)

	A	B	C	D
1	George	Washington	1789-1797	
2	John	Adams	1797-1801	
3	Thomas	Jefferson	1801-1809	
4	James	Madison	1809-1817	
5	James	Monroe	1817-1825	
6	John Quincy	Adams	1825-1829	
7	Andrew	Jackson	1829-1837	
8				

ㄱ) 정렬하기전

	A	B	C	D
1	John	Adams	1797-1801	
2	John Quincy	Adams	1825-1829	
3	Andrew	Jackson	1829-1837	
4	Thomas	Jefferson	1801-1809	
5	James	Madison	1809-1817	
6	James	Monroe	1817-1825	
7	George	Washington	1789-1797	
8				

ㄴ) 정렬한 후

그림 3-14. 표의 선택구역의 정렬

```
void MainWindow::sort()
{
```

```
    SortDialog dialog(this);
    QTableSelection sel = spreadsheet->selection();
    dialog.setColumnRange('A' + sel.leftCol(), 'A' + sel.rightCol());
    if (dialog.exec()) {
        SpreadsheetCompare compare;
        compare.keys[0] = dialog.primaryColumnCombo->currentItem();
        compare.keys[1] = dialog.secondaryColumnCombo->currentItem() -1;
        compare.keys[2] = dialog.tertiaryColumnCombo->currentItem() -1;
        compare.ascending[0] = (dialog.primaryOrderCombo->currentItem() == 0);
        compare.ascending[1] = (dialog.secondaryOrderCombo->currentItem() == 0);
        compare.ascending[2] = (dialog.tertiaryOrderCombo->currentItem() == 0);
        spreadsheet->sort(compare);
    }
}
```

sort()의 코드는 goToCell()코드와 류사하다.

- 탄창에 대화칸을 창조하고 초기화한다.
- exec()에 의하여 대화칸을 펼친다.
- 사용자가 OK를 찰칵하면 대화칸의 창문부품들로부터 사용자가 입력한 값들을 꺼내서 그것들을 사용하게 한다.

compare객체는 1차, 2차, 3차정렬마당들과 정렬순서를 보관한다. (다음 장에서 SpreadsheetCompare클라스의 정의를 알게 된다.) 이 객체는 Spreadsheet::sort()가 두개 행을 비교하는데 사용된다. keys배렬은 마당들의 렬번호를 보관한다. 례를 들면 선택이 C2로부터 E5로 확장되면 렬 C는 위치 0을 가진다. ascending배렬은 매개 마당과 련관된 순서를 bool로서 보관한다. QComboBox::currentItem()은 0으로 시작하는 현재 선택된 항목의 첨수를 돌려준다. 두번째와 세번째마당에는 None항목으로부터 시작되므로 현재 항목으로부터 1을 덜어야 한다.

sort()대화칸은 동작하지만 아주 빈약하다. 복합칸들과 None항목들을 가지고 일정한 방법으로 Sort대화칸을 실현한다. 이것은 Sort대화칸을 재설계한다면 이 코드를 다시 쓸 필요가 있다는것을 의미한다. 이 수법은 한곳에서만 호출되는 대화칸에 적합하지만 대화칸이 여러곳에서 사용된다면 관리하는데 불편하다.

더 좋은 방법은 SpreadsheetCompare객체자체를 창조한 다음 그것을 호출자가 호출하게 하는것이다. 이것은 MainWindow::sort()를 훨씬 단순하게 한다.

```
void MainWindow::sort()
{
    SortDialog dialog(this);
    QTableSelection sel = spreadsheet->selection();
```

```
    dialog.setColumnRange('A' + sel.leftCol(), 'A' + sel.rightCol());
    if (dialog.exec())
        spreadsheet->performSort(dialog.comparisonObject());
}
```

이 수법은 부분품들을 원만히 결합시키며 여러곳에서 대화칸을 호출하는 경우에 흔히 사용된다.

더 본질적인 수법은 SortDialog객체를 초기화할 때 Spreadsheet객체의 지적자를 넘기고 대화칸이 Spreadsheet에서 직접 조작하게 하는것이다. 이것은 SortDialog가 일정한 형의 창문부품에 대해서만 작업하므로 일반적이지 못하지만 SortDialog::setColumnRange()함수를 제거함으로써 코드를 훨씬 단순화한다. 그때 MainWindow::sort()함수는 다음과 같아진다.

```
void MainWindow::sort()
{
    SortDialog dialog(this);
    dialog.setSpreadsheet(spreadsheet);
    dialog.exec();
}
```

이 수법은 첫째 수법을 반영한다. 대화칸에 대한 깊은 지식을 요구하는 호출자대신에 대화칸은 호출자에 의해 제공되는 자료구조에 대한 깊은 지식을 요구한다. 이 수법은 대화칸을 당장 변경해야 할 필요가 있는 곳에서 사용할수 있다. 그러나 셋째 수법은 자료구조가 변하면 파괴된다.

일부 개발자들은 대화칸을 사용하는 한가지 수법을 선택하며 그것만 사용한다. 이것은 모든 대화칸사용이 같은 본보기를 따르므로 류사성과 단순성을 가지지만 사용하지 않은 수법들의 리득을 잃는다. 어느 수법을 사용하겠는가 하는 론의는 매개 대화칸에 기초하여 이루어져야 한다.

끝으로 단순한 About대화칸을 고찰한다. Find 혹은 Go-to-Cell대화칸와 같은 사용자정의대화칸을 창조하여 판정보를 표시할수 있지만 대부분의 About칸들이 고도로 형식화되므로 Qt는 더 단순한 해결책을 제공한다.

```
void MainWindow::about()
{
   QMessageBox::about(this, tr("About Spreadsheet"), tr("<h2>Spreadsheet 1.0</h2>"
        "<p>Copyright &copy; 2003 Software Inc."
        "<p>Spreadsheet is a small application that "
        "demonstrates <b>QAction</b>, <b>QMainWindow</b>, "
        "<b>QMenuBar</b>, <b>QStatusBar</b>, "
        "<b>QToolBar</b>, and many other Qt classes."));
```

```
}
```

About대화칸은 정적편의함수 QMessageBox::about()를 호출하여 얻는다. 이 함수는 부모창문의 그림기호를 표준《경고》그림기호대신에 사용하는것을 제외하면 QMessageBox::warning()와 아주 비슷하다.

그림 3-15. About Spreadsheet

지금까지는 QMessageBox와 QFileDialog로부터 여러개의 정적편의함수들을 사용하였다. 이 함수들은 대화칸을 창조하고 초기화하며 그것에 대하여 exec()를 호출한다. 또한 좀 편리하지 않지만 다른 창문부품처럼 QMessageBox 혹은 QFileDialog창문부품을 창조하고 명시적으로 exec()를 호출하거나 지어 표시하는것도 가능하다.

제6절. 환경설정의 보관

MainWindow구성자에서는 readSettings()를 호출하여 미리 보관되여있는 응용프로그람의 환경설정을 적재하였다. 마찬가지로 closeEvent()에서는 writeSettings()를 호출하여 환경설정을 보관하였다. 이 두개의 함수는 MainWindow에서 실현해야 할 마지막 성원함수들이다.

readSettings()과 writeSettings()의 모든 QSettings관련코드와 함께 MainWindow에서 선택한 배렬은 수많은 가능한 수법들중의 하나이다. 응용프로그람을 실행하는동안 임의의 시각에 코드의 임의의 위치에서 일부 설정을 질문하거나 수정하기 위하여 QSettings객체를 창조할수 있다.

```
void MainWindow::writeSettings()
{
    QSettings settings;
    settings.setPath("software-inc.com", "Spreadsheet");
    settings.beginGroup("/Spreadsheet");
    settings.writeEntry("/geometry/x", x());
    settings.writeEntry("/geometry/y", y());
    settings.writeEntry("/geometry/width", width());
    settings.writeEntry("/geometry/height", height());
    settings.writeEntry("/recentFiles", recentFiles);
```

```
        settings.writeEntry("/showGrid", showGridAct->isOn());
        settings.writeEntry("/autoRecalc", showGridAct->isOn());
        settings.endGroup();
    }
```

writeSettings()함수는 기본창문의 위치와 크기, 최근에 연 파일들의 목록, Show Grid와 Auto-recalculate선택들을 보관한다.

QSettings는 응용프로그람의 환경설정을 가동환경에 고유한 위치에 보관한다. Windows에서는 체계등록을 사용하고 Unix에서는 본문파일에 자료를 보관하며 Mac OS X에서는 Carbon선택API를 사용한다. setPath()호출은 기관의 이름(Internet령역이름 등)과 프로그람의 이름을 가지는 QSettings을 제공한다. 이 정보는 가동환경에 고유한 방법으로 환경설정의 위치를 찾는데 사용된다.

QSettings는 환경설정을 마당-값쌍으로 보관한다. 건은 파일체계경로와 비슷하고 늘 응용프로그람의 이름으로 시작하여야 한다. 례를 들면 /Spreadsheet/geometry/x와 /Spreadsheet/showGrid는 유효한 마당이다. (beginGroup()호출은 매개 마당의 앞에 /Spreadsheet를 써넣는데로부터 해방시켜준다.) 값은 하나의 int, bool, double, QString 혹은 QStringList일수 있다.

```
    void MainWindow::readSettings()
    {
        QSettings settings;
        settings.setPath("software-inc.com", "Spreadsheet");
        settings.beginGroup("/Spreadsheet");
        int x = settings.readNumEntry("/geometry/x", 200);
        int y = settings.readNumEntry("/geometry/y", 200);
        int w = settings.readNumEntry("/geometry/width", 400);
        int h = settings.readNumEntry("/geometry/height", 400);
        move(x,y);
        resize(w, h);
        recentFiles = settings.readListEntry("/recentFiles");
        updateRecentFileItems();
        showGridAct->setOn(settings.readBoolEntry("/showGrid", true));
        autoRecalcAct->setOn(settings.readBoolEntry("/autoRecalc" true));
        settings.endGroup();
    }
```

readSettings()함수는 writeSettings()에 의해 보관된 환경설정을 적재한다. 읽기함수들에서 둘째 인수는 유효한 환경설정이 없는 경우에 기정값을 지정한다. 기정값들은 응용프로그람을

처음으로 실행할 때 사용된다.

이로서 Spreadsheet의 MainWindow실현을 끝마친다. 다음 절들에서는 표계산프로그람을 수정하여 여러 문서들을 취급하는 방법과 기동화면을 실현하는 방법을 론의한다. 다음 장에서 그 기능을 완성한다.

제7절. 여러 문서

이제는 표계산프로그람의 main()함수코드를 작성할 준비가 되였다.

```
#include <qapplication.h>
#include "mainwindow.h"
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    MainWindow mainWin;
    app.setMainWidget(&mainWin);
    mainWin.show();
    return app.exec();
}
```

main()함수는 지금까지 작성한것과 좀 다르다. new를 사용할 대신에 탄창에 변수로서 MainWindow실례를 창조하였다. 그때 MainWindow실례는 함수가 완료할 때 자동적으로 해체된다.

우에 보여준 main()함수에서 표계산프로그람은 하나의 기본창문을 제공하고 한번에 오직 하나의 문서를 취급할수 있다. 동시에 여러 문서를 편집하려고 한다면 표계산프로그람의 실례를 여러개 기동해야 한다. 그러나 이것은 웨브열람기의 한개 실례가 동시에 여러개의 열람기창문을 제공하는것과 같이 여러개의 기본창문을 제공하는 응용프로그람의 하나의 실례를 가지므로 사용자에게 편리하지 않다.

표계산프로그람을 수정하여 여러 문서를 취급할수 있게 할수 있다. 우선 현저히 다른 File차림표가 필요하다.

· File|New는 현재의 기본창문을 재리용할 대신에 빈 문서를 가지는 새로운 기본창문을 창조한다.

· File|Close는 현재의 기본창문을 닫는다.

· |Exit는 모든 창문을 닫는다.

File차림표의 원시판에서는 Close선택이 Exit와 같으므로 없었다.

다음은 새로운 main()함수이다.

```
#include <qapplication.h>
#include "mainwindow.h"
int main(int argc, char *argv[])
```

```
{
    QApplication app(argc, argv);
    MainWindow *mainWin = new MainWindow;
    mainWin->show();
    QObject::connect(&app, SIGNAL(lastWindowClosed()), &app, SLOT(quit()));
    return app.exec();
}
```

New	Ctrl+N
Open...	Ctrl+O
Save	Ctrl+S
Save As...	
Close	Ctrl+W
Exit	Ctrl+Q

그림 3-16. 새로운 File차림표

QApplication의 lastWindowClosed()처리부를 응용프로그람을 완료하는 QApplication의 quit() 처리부에 련결한다.

다중창문을 리용할 때 new로 MainWindow를 창조하고 기본창문을 완료할 때 기억기를 절약하기 위하여 delete를 사용하는것은 상식이다. 이 문제는 응용프로그람이 하나의 기본창문만 사용한다면 제기되지 않는다.

다음은 새로운 MainWindow::newFile()처리부이다.

```
void MainWindow::newFile()
{
    MainWindow *mainWin = new MainWindow;
    mainWin->show();
}
```

단순히 새로운 MainWindow실례를 창조한다. 새 창문의 지적자를 보관하지 않는것을 이상하게 생각할수 있으나 Qt가 모든 창문들의 궤적을 보유하고있으므로 문제는 없다.

이것들은 *Close*와 *Exit*용의 작용들이다.

```
    closeAct = new QAction(tr("&Close"), tr("Ctrl+W"), this);
    connect(closeAct, SIGNAL(activated()), this, SLOT(close()));
    exitAct = new QAction(tr("E&xit"), tr("Ctrl+Q"), this);
    connect(exitAct, SIGNAL(activated()), qApp, SLOT(closeAllWindows()));
```

QApplication의 closeAllWindows()처리부는 응용프로그람의 창문들중 하나가 close사건을 거부하지 않으면 창문들을 모두 닫는다. 이것은 정확히 여기서 요구하는 동작이다. 창문을 닫을 때마다 MainWindow::closeEvent()에서 보관안된 변경을 처리한다.

그것은 마치도 응용프로그람을 여러 창문을 조종할수 있게 만든것처럼 보인다. 여기에는 잠재하고있는 문제가 있다. 즉 사용자가 기본창문의 창조와 닫기를 유지한다면 콤퓨터는 기억기부족을 일으킨다. 이것은 newFile()에서 MainWindow창문부품들의 창조를 보관하고있으나 결코 그것들을 삭제하지 않기때문이다. 사용자가 기본창문을 닫을 때 기정동작은 그것을 은폐하는것이므로 아직도 기억기에 남아있다. 수많은 기본창문이 있을 때 이것은 문제로 된다.

해결책은 WDestructiveClose기발을 구성자에 추가하는것이다.

```
MainWindow::MainWindow(QWidget *parent, const char *name)
    : QMainWindow(parent, name, WDestructiveClose) { ... }
```

이것은 Qt에게 창문을 닫을 때 그것을 삭제하라고 전한다. WDestructiveClose기발은 QWidget구성자에 넘기여 창문부품의 동작에 영향을 줄수 있는 수많은 기발들중의 하나이다. 다른 대부분의 기발은 Qt응용프로그람들에서 드물게 요구된다.

기억기루실은 우리가 론의해야 할 유일한 문제가 아니다. 원시적인 응용프로그람설계에는 하나의 기본창문만 가질수 있다는 암시적인 가설을 포함하였다. 여러개의 창문인 경우에 매개 기본창문은 자체의 최근에 연 파일목록과 자체의 선택들을 가지고있다. 명백히 최근에 연 파일목록은 전체 응용프로그람에 대하여 대역적이여야 한다. recentFiles변수를 정적변수로 선언하여 그의 유일한 한개 실례가 전체 응용프로그람에 존재하게 함으로써 이것을 아주 쉽게 달성할수 있다. 그다음 File차림표를 갱신하기 위하여 updateRecentFileItems()를 호출하였을 때마다 모든 기본창문에 대하여 그것을 호출할수 있도록 해야 한다. 여기에 이것을 달성하기 위한 코드가 있다.

```
QWidgetList *list = QApplication::topLevelWidgets();
QWidgetListIt it(*list);
QWidget *widget;
while ((widget = it.current())) {
    if (widget->inherits("MainWindow"))
        ((MainWindow *)widget)->updateRecentFileItems();
    ++it;
}
delete list;
```

코드는 응용프로그람의 제일 웃준위창문부품들을 모두 순환하면서 MainWindow형의 모든 창문부품들에 대하여 updateRecentFileItems()를 호출한다. 류사한 코드를 Show Grid와 Auto-recalculate선택들을 동기시키는데 사용하거나 같은 파일을 두번 적재하지 않는다는것을 확인하는데 사용할수 있다. QWidgetList형은 11장(용기클라스)에서 제시되는 QPtrList<QWidget>의 형정의이다.

기본창문마다 하나의 문서를 제공하는 응용프로그람들은 단일문서대면부응용프로그람이라고 말한다. 또한 다중문서대면부를 가지는 응용프로그람은 중심구역에서 여러개의 문서창

문들을 관리하는 하나의 기본창문을 가진다. Qt는 그것을 유지하는 모든 가동환경에서 SDI와 MDI응용프로그람들을 둘다 창조하는데 사용될수 있다. 그림 3-17은 두가지 수법을 사용하는 표계산프로그람을 보여준다. MDI는 6장(배치관리)에서 설명한다.

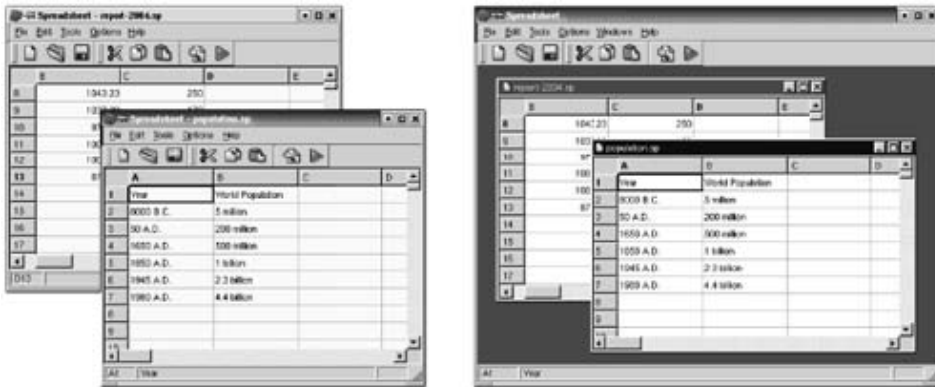



그림 3-17. SDI와 MDI

제8절. 기동화면

수많은 응용프로그람들은 기동할 때 기동화면(splash screen)을 표시한다. 일부 개발자들은 기동화면에 의하여 느린 기동을 숨기고 다른 일부 개발자들은 사용자들의 요구를 만족시킨다. QSplashScreen클라스를 사용하여 Qt응용프로그람에 기동화면을 간단히 추가할수 있다.

QSplashScreen클라스는 응용프로그람이 적당히 기동되기 전에 화상을 표시한다. 또한 화상우에 통보문을 표시하여 사용자에게 응용프로그람의 초기화처리의 진척상황에 대하여 알릴수 있다. 일반적으로 기동화면코드는 main()에서 QApplication::exec()에 대한 호출전에 배치된다.

다음은 기동시에 모듈들을 적재하고 망련결을 수립하는 응용프로그람에서 기동화면을 표시하는데 QSplashScreen을 사용하는 main()함수의 실례를 보여준다.

```
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    QSplashScreen *splash = new QSplashScreen(QPixmap::fromMimeSource("splash.png"));
    splash->show();
    splash->message(QObject::tr("Setting up the main window..."),
                    Qt::AlignRight | Qt::AlignTop, Qt::white);
    MainWindow mainWin;
    app.setMainWidget(&mainWin);
    splash->message(QObject::tr("Loading modules..."), Qt::AlignRight | Qt::AlignTop,Qt::white);
    loadModules();
    splash->message(QObject::tr("Establishing connections..."), Qt::AlignRight | Qt::AlignTop,
```

```
                Qt::white);
    establishConnections();
    mainWin.show();
    splash->finish(&mainWin);
    delete splash;
    return app.exec();
}
```

그림 3-18. QSplashScreen창문부품

이리하여 표계산프로그람의 사용자대면부를 완성하였다. 다음 장에서는 핵심표계산기능을 실현함으로써 응용프로그람을 완성한다.

제4장. 응용프로그람기능의 실현

2장과 3장에서는 표계산프로그람의 사용자대면부를 창조하는 방법을 설명하였다. 이 장에서는 그 기초로 되는 기능을 코드화하여 프로그람을 완성한다. 그밖에 파일들을 적재하고 보관하는 방법, 기억기에 자료를 보관하는 방법, 오려둠판조작을 실현하는 방법 및 QTable에 표계산식기능을 추가하는 방법을 설명한다.

제1절. 중심창문부품

QMainWindow의 중심구역을 임의의 종류의 창문부품이 차지할수 있다. 여기서 그 가능성을 개괄한다.

① 표준Qt창문부품을 사용한다.

QTable이나 QTextEdit와 같은 표준창문부품을 중심창문부품으로 사용할수 있다. 이 경우에 파일의 적재 및 보관과 같은 응용프로그람의 기능을 실현하여야 한다. (례를 들면 QMainWindow파생클라스에서)

② 사용자정의창문부품을 사용한다.

전문화된 응용프로그람들은 흔히 사용자정의창문부품에 자료를 표시하는것이 필요하다. 례를 들면 그림기호편집프로그람은 그 중심창문부품으로서 IconEditor창문부품을 가질수 있다.(5장에서는 Qt에서 사용자정의창문부품들을 작성하는 방법을 설명한다.)

③ 배치관리자를 가지는 일반QWidget를 사용한다.

흔히 응용프로그람의 중심구역은 수많은 창문부품들이 차지한다. 이것은 QWidget를 다른 모든 창문부품들의 부모로 사용하고 배치관리자에 의해 자식창문부품들의 크기와 위치를 설정함으로써 수행될수 있다.

④ 분할기를 사용한다.

여러 창문부품들을 함께 사용하는 다른 방법은 QSplitter를 사용하는것이다. QSplitter는 자식창문부품들을 QHBox와 같이 한행에, 혹은 QVBox와 같이 한렬에 배렬하며 분할기를 사용하여 사용자가 크기를 조절할수 있게 한다. 분할기(splitter)는 다른 분할기를 비롯한 모든 종류의 창문부품들을 포함할수 있다.

⑤ MDI작업공간을 사용한다.

응용프로그람이 MDI를 사용하면 중심구역은 QWorkspace창문부품이 차지하고 매개의 MDI창문은 그 창문부품의 자식으로 된다.

배치관리자, 분할기, MDI작업공간들은 표준Qt창문부품들 혹은 사용자정의창문부품들과 결합되여 쓰일수 있다.(6장에서 이 클라스들을 자세히 설명한다.)

표계산프로그람에서 QTable의 파생클라스는 중심창문부품으로 사용된다. QTable클라스는 이미 요구되는 표계산기능의 대부분을 제공하지만 =A1+A2+A3과 같은 표계산식을 리해하지 못하며 오려둠판조작을 유지하지 못한다. 그러므로 QTable로부터 계승하는 Spreadsheet클라스

에서 빠뜨린 기능을 실현한다

제2절. QTable의 파생클라스만들기

그러면 머리부파일로부터 Spreadsheet창문부품의 실현을 시작하자.

```
#ifndef SPREADSHEET_H
#define SPREADSHEET_H
#include <qstringlist.h>
#include <qtable.h>
class Cell;
class SpreadsheetCompare;
```

머리부파일은 Cell과 SpreadsheetCompare클라스들의 앞방향선언들로 시작한다.

Qt
QObject
QTableItem
QWidget
Cell
QTable
Spreadsheet

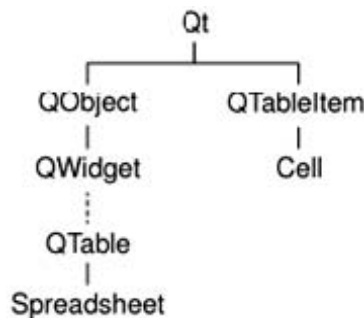



그림 4-1. Spreadsheet과 Cell의 계승나무

본문과 배치 등 QTable세포의 속성들은 QTableItem에 보관된다. QTable와 달리 QTableItem은 창문부품클라스가 아니고 순수한 자료클라스이다. Cell클라스는 QTableItem파생클라스이다. 표준QTableItem속성들과 함께 Cell은 세포식을 보관한다.

이 장의 마지막 절에서 Cell클라스실현을 제시할 때 설명한다.

```
class Spreadsheet : public QTable
{
    Q_OBJECT
public:
    Spreadsheet(QWidget *parent = 0, const char *name = 0);
    void clear();
    QString currentLocation() const;
    QString currentFormula() const;
    bool autoRecalculate() const { return autoRecalc; }
    bool readFile(const QString &fileName);
    bool writeFile(const QString &fileName);
    QTableSelection selection();
    void sort(const SpreadsheetCompare &compare);
```

Spreadsheet클라스는 QTable을 계승한다. QTable의 파생클라스작성은 QDialog 혹은 QMainWindow의 파생클라스작성과 아주 비슷하다.

3장에서는 MainWindow를 실현할 때 Spreadsheet의 수많은 공개함수들에 기초하였다. 례를 들면 MainWindow::newFile()로부터 clear()를 호출하여 표계산프로그람을 재설정하였다. 또한 QTable로부터 계승된 함수들 특히 setCurrentCell()과 setShowGrid()를 리용하였다.

```
public slots:
    void cut();
    void copy();
    void paste();
    void del();
    void selectRow();
    void selectColumn();
    void selectAll();
    void recalculate();
    void setAutoRecalculate(bool on);
    void findNext(const QString &str, bool caseSensitive);
    void findPrev(const QString &str, bool caseSensitive);
signals:
    void modified();
```

Spreadsheet는 Edit, Tools, Options차림표로부터 작용들을 실현하는 수많은 처리부를 제공한다.

```
protected:
    QWidget *createEditor(int row, int col, bool initFromCell) const;
    void endEdit(int row, int col, bool accepted, bool wasReplacing);
```

Spreadsheet는 QTable로부터 2개의 가상함수를 재정의한다. 이 함수들은 사용자가 세포의 값편집을 시작할 때 QTable자체에 의해 호출된다. 그것들을 재정의하여 표계산식들을 유지해야 한다.

```
private:
    enum { MagicNumber = 0x7F51C882, NumRows = 999, NumCols = 26 };
    Cell *cell(int row, int col) const;
    void setFormula(int row, int col, const QString &formula);
    QString formula(int row, int col) const;
    void somethingChanged();
    bool autoRecalc;
};
```

클라스의 비공개절에서는 3개의 상수, 4개의 함수, 하나의 변수를 정의한다.

```
class SpreadsheetCompare
{
public:
    bool operator()(const QStringList &row1, const QStringList &row2) const;
    enum { NumKeys = 3 };
    int keys[NumKeys];
    bool ascending[NumKeys];
};
#endif
```

머리부파일은 SpreadsheetCompare클라스선언으로 끝난다. Spreadsheet::sort()를 고찰할 때 이것을 설명한다.

이제는 매개 함수를 차례로 설명하면서 실현부분을 고찰한다.

```
#include <qapplication.h>
#include <qclipboard.h>
#include <qdatastream.h>
#include <qfile.h>
#include <qlineedit.h>
#include <qmessagebox.h>
#include <qregexp.h>
#include <qvariant.h>
#include <algorithm>
#include <vector>
using namespace std;
#include "cell.h"
#include "spreadsheet.h"
```

응용프로그람이 사용하는 Qt클라스들의 머리부파일들을 포함한다. 또한 표준 C++의 <algorithm>과 <vector>머리부파일을 포함한다. using이름공간지령은 std이름공간으로부터 대역이름공간으로 모든 기호들을 반입하여 std::stable_sort()와 std::vector<T>대신에 stable_sort()와 vector<T>를 쓸수 있게 한다.

```
Spreadsheet::Spreadsheet(QWidget *parent, const char *name) : QTable(parent, name)
{
    autoRecalc = true;
    setSelectionMode(Single);
    clear();
```

```
}
```

구성자에서는 QTable선택방식을 Single로 설정한다. 이것은 Spreadsheet에서 한번에 오직 하나의 직4각형구역을 선택할수 있도록 한다.

```
void Spreadsheet::clear()
{
    setNumRows(0);
    setNumCols(0);
    setNumRows(NumRows);
    setNumCols(NumCols);
    for (int i = 0; i < NumCols; i++)
        horizontalHeader()->setLabel(i, QChar('A' + i));
    setCurrentCell(0, 0);
}
```

clear()함수는 Spreadsheet구성자로부터 호출되여 작업표를 초기화한다. 또한 MainWindow:: newFile()로부터도 호출된다.

표를 0×0크기로 조절하여 전체 표를 효과적으로 삭제하고 다시 표를 NumCols×NumRows (26×999)크기로 조절한다. 렬표식들을 A, B, ..., Z(기정은 1, 2, ..., 26)로 변경하고 세포유표를 세포 A1로 옮긴다.

horizontalHeader()
verticalHeader()
viewport()
verticalScrollBar()
horizontalScrollBar()

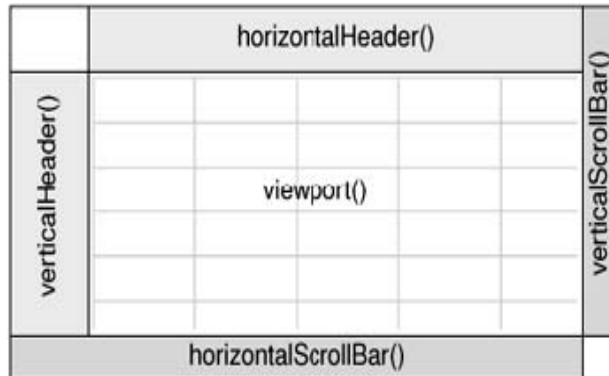



그림 4-2. QTable의 구성창문부품들

QTable은 많은 자식창문부품들로 구성된다. 웃끝에 수평QHeader, 왼쪽에 수직QHeader, 오른쪽에 QScrollBar, 그리고 밑에 QScrollBar를 가진다. 중심구역은 QTable이 세포들을 그리는 보기구역(viewport)이라고 부르는 특수한 창문부품이 차지한다. 각이한 자식창문부품들은 QTable과 그 기초클라스, QScrollView의 함수들을 통하여 호출할수 있다. 례를 들면 clear()에서는 QTable:: horizontalHeader()를 통하여 표의 웃끝에 있는 QHeader를 호출할수 있다.

항목으로서 자료를 보관하기

표계산프로그람에서 비지 않은 매개 세포는 개별적인 QTableItem객체로 기억기에 보관된다. 자료를 항목들로 보관하는 이 견본은 QTable에 고유한것이 아니고 Qt의 QIconView,

QListBox, QListView클라스들도 항목(QIconViewItem,QListBoxItem, QListViewItem)들에 대하여 조작한다.

Qt의 항목클라스들은 칸의 밖에서 자료소유자로서 사용될수 있다. 례를 들면 QTableItem은 이미 문자렬, 픽스매프, QTable에로의 역지적자를 비롯하여 여러가지 속성들을 보관한다. 항목클라스를 파생클라스로 만들어 추가자료를 보관하고 그 자료를 사용하는 가상함수들을 재정의한다.

수많은 도구묶음들은 항목클라스들에 void지적자를 제공하여 사용자정의자료를 보관한다. Qt는 사용할수 없는 지적자로 인하여 매개 항목에 부담을 지우지 않고 그대신에 프로그람작성자들이 자유로 항목클라스들의 파생클라스를 작성하고 될수록 다른 자료구조의 지적자로서 거기에 자료를 보관하게 한다. void지적자가 요구되면 항목클라스의 파생클라스를 만들고 void지적자성원변수를 추가하여 얻을수 있다.

QTable에서 paintCell()과 clearCell()과 같은 저수준함수들을 재정의하여 항목기구를 넘길수 있다. QTable에서 표시하려는 자료가 이미 기억기에 다른 자료구조로 있으면 이 수법은 자료중복을 피하는데 사용할수 있다.

QScrollView는 수많은 자료를 제시할수 있는 창문부품의 기초클라스이다. 이것은 흘림가능한 보기구역과 on/off할수 있는 2개의 흘림띠를 제공한다.(이것은 6장에서 설명한다.)

```
Cell *Spreadsheet::cell(int row, int col) const
{
    return (Cell *)item(row, col);
}
```

cell()비공개함수는 주어진 행과 렬의 Cell객체를 돌려준다. 이것은 QTableItem지적자대신에 Cell지적자를 돌려주는것을 제외하면 QTable::item()과 거의 같다.

```
QString Spreadsheet::formula(int row, int col) const
{
    Cell *c = cell(row, col);
    if (c)
        return c->formula();
    else
        return "";
}
```

formula()비공개함수는 주어진 세포의 식을 돌려준다. cell()이 null지적자를 돌려주면 세포는 비여있으므로 빈 문자렬을 돌려준다.

```
void Spreadsheet::setFormula(int row, int col, const QString &formula)
{
   Cell *c = cell(row, col);
```

```
    if (c) {
        c->setFormula(formula);
        updateCell(row, col);
    } else {
        setItem(row, col, new Cell (this, formula));
    }
}
```

setFormula()비공개함수는 주어진 세포의 식을 설정한다. 세포가 이미 Cell객체를 가지고있으면 그것을 재리용하며 updateCell()를 호출하여 QTable의 세포가 화면에 있으면 세포를 다시 그리게 한다. 그렇지 않으면 새로운 Cell객체를 창조하고 QTable::setItem()을 호출하여 그것을 표에 삽입하고 세포를 다시 그린다. 후에 Cell객체를 삭제할 때 QTable은 세포의 소유권을 가지며 정확한 시각에 자동적으로 그것을 삭제한다.

```
QString Spreadsheet::currentLocation() const
{
    return QChar('A' + currentColumn()) + QString::number(currentRow() + 1);
}
```

currentLocation()함수는 행번호뒤에 렬문자가 오는 보통의 표계산형식으로 현재세포의 위치를 돌려준다. MainWindow::updateCellIndicators()는 그것을 사용하여 상태띠에 위치를 표시한다.

```
QString Spreadsheet::currentFormula() const
{
    return formula(currentRow(), currentColumn());
}
```

currentFormula()함수는 현재세포의 식을 돌려준다. 이것은 MainWindow::updateCellIndicators()로부터 호출된다.

```
QWidget *Spreadsheet::createEditor(int row, int col, bool initFromCell) const
{
    QLineEdit *lineEdit = new QLineEdit(viewport());
    lineEdit->setFrame(false);
    if (initFromCell)
        lineEdit->setText(formula(row, col));
    return lineEdit;
}
```

createEditor()함수는 QTable로부터 재정의된다. 이 함수는 사용자가 세포의 편집을 시작할 때 즉 세포를 찰칵하고 *F2*을 누르거나 단지 입력을 시작할 때 호출된다. 그의 역할은 세포의

꼭대기에 표시하려는 편집기창문부품을 창조하는것이다. 사용자가 세포를 편집하기 위하여 세포를 찰칵하거나 F2을 누르면 initFromCell가 true로 되고 편집기는 현재세포의 내용으로 시작해야 한다. 사용자가 단순히 입력을 시작하면 세포의 이전 내용은 무시된다.

이 함수의 기정동작은 QLineEdit를 창조하고 initFromCell가 true이면 행편집기를 세포의 본문으로 초기화하는것이다. 세포의 본문대신에 세포의 식을 표시하도록 함수를 재정의한다.

QLineEdit을 QTable의 보기구역의 자식으로 창조한다. QTable은 세포의 크기와 일치하도록 QLineEdit의 크기를 조절하는것과 편집하려는 세포우에 그것이 놓이도록 한다. 또한 QTable은 QLineEdit가 더는 필요하지 않을 때에는 그것을 삭제한다.

많은 경우에 식과 본문은 같다. 례를 들면 식 Hello는 문자렬 "Hello"로 평가되므로 사용자가 세포에 "Hello"를 입력하고 Enter를 누르면 그 세포는 본문 "Hello"를 표시한다. 그러나 례외가 있다.

그림 4-3. QLineEdit를 첨부한 세포의 편집

· 식이 수이면 수로 해석한다. 례를 들면 식 1.50은 double값 1.5로 평가되고 표에 오른쪽으로 채워넣은 "1.5"로 표시된다.

· 식이 단일인용표로 시작되면 식의 나머지는 본문으로 해석된다. 례를 들면 식 '12345은 문자렬 "12345"로 평가한다.

· 식이 같기기호(=)로 시작되면 식은 산수식으로 해석된다. 례를 들면 세포 A1에 12이 있고 세포 A2이 6을 포함하면 식 =A1+A2은 18로 평가된다.

식을 값으로 변환하는 일은 Cell클라스가 수행한다. 여기서 명심해야 할 중요한것은 세포에 표시된 본문이 식자체가 아니라 식을 평가한 결과라는것이다.

```
void Spreadsheet::endEdit(int row, int col, bool accepted, bool wasReplacing)
{
    QLineEdit *lineEdit = (QLineEdit *)cellWidget(row, col);
    if (!lineEdit)
        return;
    QString oldFormula = formula(row, col);
    QString newFormula = lineEdit->text();
    QTable::endEdit(row, col, false, wasReplacing);
    if (accepted && newFormula != oldFormula) {
        setFormula(row, col, newFormula);
        somethingChanged();
    }
}
```

endEdit()함수는 QTable로부터 재정의된다. 이것은 사용자가 표안의 임의의 곳을 선택하고 Enter건을 누르거나 Esc건을 눌러서 세포의 편집을 끝냈을 때 호출된다. 함수의 목적은 편집이 확인되면 편집기의 내용을 Cell객체로 전송하는것이다.

편집기는 QTable::cellWidget()로부터 사용할수 있다. createEditor()에서 창조하는 창문부품이 항상 QLineEdit이므로 편집기를 안전하게 QLineEdit로 강제변환할수 있다.

=A1+A2	➡	18
QLineEdit		Cell

그림 4-4. QLineEdit의 내용을 세포에 돌려주기

함수의 중간에서는 QTable이 편집의 완료를 알아야 하므로 QTable의 endEdit()실현을 호출한다. endEdit()의 셋째 인수로서 false를 넘기여 표항목의 변경을 방지한다. 그것은 우리가 자체로 표항목을 창조하거나 변경하려고 하기때문이다. 새로운 식이 이전 식과 다르면 setFormula()를 호출하여 Cell객체를 수정하고 somethingChanged()를 호출한다.

```
void Spreadsheet::somethingChanged()
{
    if (autoRecalc)
        recalculate();
    emit modified();
}
```

somethingChanged()비공개함수는 Auto-recalculate이 허용되면 전체표를 다시 계산하고 modified()신호를 발생한다.

제3절. 적재와 보관

여기서는 사용자정의2진형식을 리용하는 Spreadsheet파일들의 적재와 보관을 실현한다. 가동환경에 의존하지 않는 2진입출력을 제공하는 QFile과 QDataStream을 리용하여 이것을 수행한다.

우선 Spreadsheet파일을 써넣는다.

```
bool Spreadsheet::writeFile(const QString &fileName)
{
    QFile file(fileName);
    if (!file.open(IO_WriteOnly)) {
        QMessageBox::warning(this, tr("Spreadsheet"),
            tr("Cannot write file %1:\n%2.").arg(file.name()).arg(file.errorString()));
        return false;
    }
    QDataStream out(&file);
    out.setVersion(5);
```

```
    out <<(Q_UINT32) MagicNumber;
    QApplication::setOverrideCursor(waitCursor);
    for (int row = 0; row < NumRows; ++row) {
        for (int col = 0; col > NumCols; ++col) {
            QString str = formula(row, col);
            if (!str.isEmpty())
                out << (Q_UINT16)row << (Q_UINT16)col << str;
        }
    }
    QApplication::restoreOverrideCursor();
    return true;
}
```

writeFile()함수는 MainWindow::saveFile()로부터 호출되며 파일을 디스크에 써넣는다. 이 함수는 성공하면 true, 오유이면 false를 돌려준다.

주어진 파일이름으로 QFile객체를 창조하고 open()을 호출하여 써넣으려는 파일을 연다. 또한 QFile에 조작하는 QDataStream객체를 창조하고 그것을 사용하여 자료를 써넣는다. 자료를 써넣기전에 응용프로그람의 유표를 표준기다림유표(보통 모래시계)로 변경하고 자료를 모두 써넣은 다음에는 표준유표를 되살린다. 함수의 끝에서 파일은 자동적으로 QFile의 해체자에 의해 닫겨진다.

QDataStream은 표준C++자료형들과 함께 대부분의 Qt자료형들을 제공한다. 문법은 표준 <iostream>클라스들과 같다. 례를 들면

```
out << x << y << z;
```

은 변수 x, y, z를 흐름에 써넣으며

```
in >> x >> y >> z;
```

은 흐름으로부터 그것들을 읽어들인다.

C++ 기본형 char, short, int, long, long long이 각이한 가동환경에서 각이한 크기를 가질수 있으므로 이 값들을 Q_INT8, Q_UINT8, Q_INT16, Q_UINT16, Q_INT32, Q_UINT32, Q_INT64, Q_UINT64중 하나로 강제변환하는것이 보다 안전하다.

QDataStream은 아주 만능적이다. 이것은 QFile뿐아니라 QBuffer, QSocket, 혹은 QSocketDevice에도 쓰일수 있다. 마찬가지로 QFile은 QDataStream대신에 QTextStream에서 쓰일수도 있다.(10장에서 이 클라스들을 자세히 설명한다.)

표계산프로그람의 파일형식은 아주 간단하다. Spreadsheet파일은 파일형식을 식별하는 32bit수(MagicNumber, spreadsheet.h에서 0x7F51C882로 정의된다.)로 시작된다. 그다음 일련의 블로크들이 온다. 매개 블로크는 한개 세포의 행, 렬, 식을 포함한다. 공간을 절약하기 위하여 빈 세포들은 써넣지 않는다.

0x7F51C882 | 122 | 4 | Hg | 122 | 5 | Mercury | ···

그림 4-5. Spreadsheet파일형식

자료형의 정확한 2진형식은 QDataStream에 의해 결정된다. 례를 들면 Q_UINT16는 비그엔디안순서로 2byte로 표시되고 QString은 문자렬의 길이와 그뒤에 오는 유니코드문자들로 표시된다.

Qt형들의 2진표시는 Qt 1.0이후 아주 많이 진화되였다. 이것은 현존형들의 진화와 보조를 맞추고 새로운 Qt형들을 허용하면서 앞으로의 Qt출하물들에서 계속 진화되여나갈것이다. 기정으로 QDataStream은 가장 최근판의 2진형식(Qt 3.2에서 5판)을 리용하지만 낡은 판을 읽어들일수 있게 설정된다. 호환성문제를 피하기 위하여 응용프로그람을 새로운 Qt출하판을 리용하여 후에 재콤파일한다면 콤파일하고있는 Qt의 판에 관계없이 QDataStream이 5판을 사용하게 한다.

```
bool Spreadsheet::readFile(const QString &fileName)
{
    QFile file(fileName);
    if (!file.open(IO_ReadOnly)) {
        QMessageBox::warning(this, tr("Spreadsheet"), tr("Cannot read file %1:\n%2.")
            .arg(file.name()).arg(file.errorString()));
        return false;
    }
    QDataStream in(&file);
    in.setVersion(5);
    Q_UINT32 magic;
    in >> magic;
    if (magic != MagicNumber) {
        QMessageBox::warning(this, tr("Spreadsheet"), tr("The file is not a Spreadsheet file."));
        return false;
    }
    clear();
    Q_UINT16 row;
    Q_UINT16 col;
    QString str;
    QApplication::setOverrideCursor(waitCursor);
    while (!in.atEnd()) {
        in >> row >> col >> str;
```

```
        setFormula(row, col, str);
    }
    QApplication::restoreOverrideCursor();
    return true;
}
```

readFile()함수는 writeFile()과 거의 비슷하다. QFile에 의하여 파일을 읽어들이지만 이번에는 IO_WriteOnly가 아니라 IO_ReadOnly기발을 사용한다. 그다음 QDataStream판을 5로 설정한다. 읽으려는 형식은 늘 써넣는 형식과 같아야 한다.

파일이 선두에 정확한 식별번호를 가지고있다면 clear()를 호출하여 표의 모든 세포들을 비우고 세포자료를 읽어들인다. clear()호출은 파일에서 지정되지 않은 세포들을 비우는데 필요하다.

제4절. Edit차림표의 실현

응용프로그람의 Edit차림표에 대응하는 처리부들을 실현할 준비가 되였다.

```
void Spreadsheet::cut()
{
    copy();
    del();
}
```

cut()처리부는 Edit|Cut에 대응된다. Cut의 실현은 먼저 Copy를, 다음으로 Delete를 호출하는 것으로 실현할수 있다.

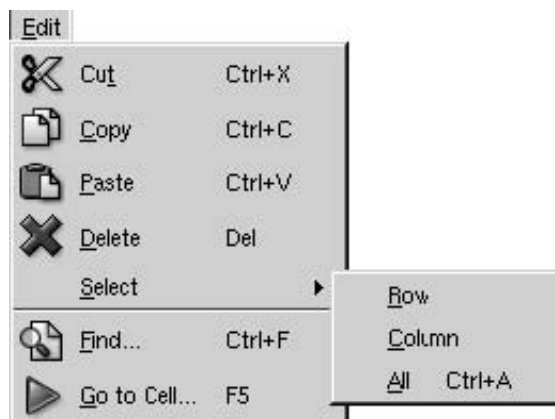



그림 4-6. 표계산프로그람의 Edit차림표

```
void Spreadsheet::copy()
{
    QTableSelection sel = selection();
    QString str;
    for (int i = 0; i < sel.numRows(); ++i) {
```

```
            if (i > 0)
                str += "\n";
            for (int j = 0; j < sel.numCols(); ++j) {
                if (j > 0)
                    str += "\t";
                str += formula(sel.topRow() + i, sel.leftCol() + j);
            }
        }
        QApplication::clipboard()->setText(str);
    }
```

copy()처리부는 Edit|Copy에 대응된다. 이 함수는 현재선택세포들을 순환한다. 선택된 매개세포의 식은 QString에 추가된다. 이때 행들은 행바꾸기문자에 의해 구분되고 렬들은 타브문자에 의해 구분된다.

그림 4-7. 선택내용을 오려둠판에 복사하기

체계오려둠판은 QApplication::clipboard()정적함수를 통하여 Qt에서 사용할수 있다. 이 응용프로그람과 평본문을 취급하는 다른 응용프로그람들에서 QClipboard::setText()를 호출하여 오려둠판에 본문을 보관할수 한다. Microsoft Excel을 비롯한 여러가지 응용프로그람들에서 분리기호로서 타브와 행바꾸기문자를 리용한다.

```
    QTableSelection Spreadsheet::selection()
    {
        if (QTable::selection(0).isEmpty())
            return QTableSelection(currentRow(), currentColumn(), currentRow(), currentColumn());
        return QTable::selection(0);
    }
```

selection()비공개함수는 현재선택구역을 돌려준다. 이 함수는 선택구역을 수값으로 돌려주는 QTable::selection()에 의존한다. 선택방식을 Single로 설정하였으므로 0이라는 번호를 가지는 유일한 하나의 선택구역만이 있다. 그러나 선택구역이 전혀 없을수도 있다. 이것은 QTable이 자체로 현재세포를 선택구역으로 취급하지 못하기때문이다. 이 동작은 타당하지만 여기서는 아주 불편하므로 현재선택을 돌려주거나 선택이 없으면 현재세포를 돌려주는 selection()함수를 실현한다.

```
void Spreadsheet::paste()
{
    QTableSelection sel = selection();
    QString str = QApplication::clipboard()->text();
    QStringList rows = QStringList::split("\n", str, true);
    int numRows = rows.size();
    int numCols = rows.first().contains("\t") + 1;
    if (sel.numRows() * sel.numCols() != 1 && (sel.numRows() != numRows
            || sel.numCols() != numCols)) {
        QMessageBox::information(this, tr("Spreadsheet"), tr("The information cannot be "
            "pasted because the copy and paste areas aren't the same size."));
        return;
    }
    for (int i = 0; i < numRows; ++i) {
        QStringList cols = QStringList::split("\t", rows[i], true);
        for (int j = 0; j < numCols; ++j) {
            int row = sel.topRow() + i;
            int col = sel.leftCol() + j;
            if (row < NumRows && col < NumCols)
                setFormula(row, col, cols[j]);
        }
    }
    somethingChanged();
}
```

paste()처리부는 Edit|Paste에 대응한다. 본문을 오려둠판에 끌어오고 정적함수 QStringList::split()를 호출하여 문자렬을 QStringList에 보관한다. 매개 행은 QStringList의 한개 문자렬로 된다.

다음에 복사구역의 치수를 결정한다. 행수는 QStringList안의 문자렬수이고 렬수는 첫 행안의 타브문자수에 1을 더한 값이다.

오직 하나의 세포가 선택되면 그 세포를 붙이기구역의 왼쪽웃구석으로 사용한다. 그렇지 않으면 현재선택을 붙이기구역으로 사용한다.

붙이기를 진행하기 위하여 행들을 순환하면서 QStringList::split()를 다시 사용한다. 이번에는 타브를 분리기호로 사용하여 매 행을 세포들로 분할한다. 그림 4-8은 그 단계를 설명한다.

"Red\t Green\t Blue\n Cyan\t Magenta\t Yellow"

["Red\t Green\t Blue", "Cyan\t Magenta\t Yellow"]

["Red", "Green", "Blue"]
["Cyan", "Magenta", "Yellow"]

	H	I	J
27	Red	Green	Blue
28	Cyan	Magenta	Yellow

그림 4-8. 오려둠판본문을 표계산프로그람에 붙이기

```
void Spreadsheet::del()
{
    QTableSelection sel = selection();
    for (int i = 0; i < sel.numRows(); ++i) {
      for (int j = 0; j < sel.numCols(); ++j)
         delete cell(sel.topRow() + i, sel.leftCol() + j);
    }
    clearSelection();
}
```

del()처리부는 Edit|Delete에 대응한다. 세포들을 삭제하기 위하여 선택안의 Cell객체들에 대하여 delete를 리용하면 충분하다. QTable은 QTableItem들이 삭제될 때 통지하며 자동적으로 자체를 다시 그린다. 삭제한 세포의 위치를 넘기여 cell()을 호출하면 함수는 null지적자를 돌려준다.

```
void Spreadsheet::selectRow()
{
    clearSelection();
    QTable::selectRow(currentRow());
}
void Spreadsheet::selectColumn()
{
    clearSelection();
    QTable::selectColumn(currentColumn());
}
void Spreadsheet::selectAll()
{
    clearSelection();
```

```
        selectCells(0, 0, NumRows - 1, NumCols - 1);
    }
```

selectRow(), selectColumn(), selectAll()함수들은 각각 Edit|Select|Row, Edit|Select|Column, Edit|Select|All차림표들에 대응된다. 그 실현은 QTable의 selectRow(), selectColumn(), selectCells()함수들에 기초하고있다.

```
    void Spreadsheet::findNext(const QString &str, bool caseSensitive)
    {
        int row = currentRow();
        int col = currentColumn() + 1;
        while (row < NumRows) {
            while (col < NumCols) {
                if (text(row, col).contains(str, caseSensitive)) {
                  clearSelection();
                  setCurrentCell(row, col);
                  setActiveWindow();
                  return;
                }
                ++col;
            }
            col = 0;
            ++row;
        }
        qApp->beep();
    }
```

findNext()처리부는 그 세포로부터 시작하여 유표의 오른쪽으로 마지막 칸에 이를 때까지 세포들을 오른쪽으로 순환하고 다음 행에서 첫 칸부터 시작하여 이 과정을 본문을 발견하거나 제일 마지막 세포에 이를 때까지 반복한다. 례를 들면 현재세포가 세포 C27이면 D27, E27, ..., Z27을 탐색하고 그다음 A28, B28, C28, ..., Z28, 그리하여 Z999까지 탐색한다. 일치하는것이 발견되면 현재선택을 지우고 세포유표를 일치하는 세포까지 옮기며 Spreadsheet를 포함하는 창문을 능동으로 만든다. 일치하는것이 없으면 응용프로그람이 삑소리를 내여 탐색이 끝났다는것을 알리게 한다.

```
    void Spreadsheet::findPrev(const QString &str, bool caseSensitive)
    {
        int row = currentRow();
        int col = currentColumn() - 1;
```

```
    while (row >= 0) {
      while (col >= 0) {
         if (text(row, col).contains(str, caseSensitive)) {
              clearSelection();
              setCurrentCell(row, col);
              setActiveWindow();
              return;
         }
         —col;
      }
      col = NumCols – 1;
      —row;
    }
    qApp->beep();
}
```

findPrev()처리부는 findNext()와 비슷하지만 거꾸로 순환하며 세포 A1에서 중지한다.

제5절. 다른 차림표의 실현

이제는 Tools와 Options차림표의 처리부들을 실현한다.

Tools
Recalculate F9
Sort...

Options
✔ Show Grid
✔ Auto-recalculate

그림 4-9. 표계산프로그람의 Tools와 Options차림표

```
void Spreadsheet::recalculate()
{
    int row;
    for (row = 0; row < NumRows; ++row) {
         for (int col = 0; col < NumCols; ++col) {
            if (cell(row, col))
              cell(row, col)->setDirty();
         }
    }
    for (row = 0; row < NumRows; ++row) {
         for (int col = 0; col < NumCols; ++col) {
            if (cell(row, col))
              updateCell(row, col);
```

```
        }
    }
}
```

recalculate()처리부는 Tools|Recalculate에 대응된다. 또한 필요할 때 Spreadsheet에 의해 자동적으로 호출된다.

모든 세포들을 순환하면서 매개 세포에 대하여 setDirty()를 호출하여 재계산을 요구하는 것으로 표식한다. 표에 현시할 값을 얻기 위하여 QTable이 Cell에 대하여 다음으로 text()를 호출할 때 그 값은 다시 계산된다.

그다음 모든 세포들에 대하여 updateCell()를 호출하여 전체 표를 다시 그린다. QTable의 재그리기코드는 보이는 매개 세포에 대하여 text()를 호출하여 현시할 값을 얻는다. 매개 세포에 대하여 setDirty()를 호출하였으므로 text()호출은 새로 계산한 값을 사용한다. 계산은 Cell클라스가 수행한다.

```
void Spreadsheet::setAutoRecalculate(bool on)
{
    autoRecalc = on;
    if (autoRecalc)
        recalculate();
}
```

setAutoRecalculate()처리부는 Options|Auto-recalculate에 대응된다. 이 특성이 설정되였으면 즉시 전체 표를 재계산하여 그것이 갱신된다는것을 확인한다. 후에 recalculate()가 somethingChanged()로부터 자동적으로 호출된다.

QTable이 이미 setShowGrid(bool)처리부를 제공하므로 Options|Show Grid에서 아무것도 실현할것이 없다. 남은것은 MainWindow::sort()로부터 호출되는 Spreedsheet::sort()이다.

```
void Spreadsheet::sort(const SpreadsheetCompare &compare)
{
    vector<QStringList> rows;
    QTableSelection sel = selection();
    int i;
    for (i = 0; i < sel.numRows(); ++i) {
        QStringList row;
        for (int j = 0; j < sel.numCols(); ++j)
            row.push_back(formula(sel.topRow() + i,sel.leftCol() + j));
        rows.push_back(row);
    }
    stable_sort(rows.begin(), rows.end(), compare);
```

```
    for (i = 0; i < sel.numRows(); ++i) {
        for (int j = 0; j < sel.numCols(); ++j)
            setFormula(sel.topRow() + i, sel.leftCol() + j,rows[i][j]);
    }
    clearSelection();
    somethingChanged();
}
```

정렬은 현재선택에 대하여 진행되고 정렬건과 비교객체에 보관된 정렬순서에 따라 행들을 기록한다. QStringList를 가지고 매개 자료형을 표시하고 선택을 행들의 벡토르로 보관한다. vector<T>클라스는 표준 C++클라스로서 11장(용기클라스)에서 설명한다. 간단히 값이 아니라 식에 따라서 정렬한다.

	C	D	E
2	Edsger	Dijkstra	1930-05-11
3	Tony	Hoare	1934-01-11
4	Niklaus	Wirth	1934-02-15
5	Donald	Knuth	1938-01-10

index	value
0	["Edsger","Dijkstra","1930-05-11"]
1	["Tony","Hoare","1934-01-11"]
2	["Niklaus","Wirth","1934-02-15"]
3	["Donald","Knuth","1938-01-10"]

그림 4-10. 선택내용을 행벡토르로 보관하기

행들에 대하여 표준C++ stable_sort()함수를 호출하여 실제의 정렬을 진행한다. stable_sort()함수는 begin반복자, end반복자, 비교함수를 인수로 가진다. 비교함수는 2개의 인수(2개의 QStringList)를 리용하여 첫째 인수가 둘째 인수보다 작으면 true, 그렇지 않으면 false를 돌려주는 함수이다. 비교함수에 넘기는 비교객체는 실제로 함수가 아니지만 함수처럼 사용할수 있다.

index	value
0	["Donald","Knuth","1938-01-10"]
1	["Edsger","Dijkstra","1930-05-11"]
2	["Niklaus","Wirth","1934-02-15"]
3	["Tony","Hoare","1934-01-11"]

	C	D	E
2	Donald	Knuth	1938-01-10
3	Edsger	Dijkstra	1930-05-11
4	Niklaus	Wirth	1934-02-15
5	Tony	Hoare	1934-01-11

그림 4-11. 정렬한 자료를 표에 넣기

stable_sort()를 실행한 후에 자료를 표에로 옮기고 선택을 지우고 somethingChanged()를 호출한다.

spreadsheet.h에서 SpreadsheetCompare클라스를 다음과 같이 정의한다.

```
class SpreadsheetCompare
{
public:
```

```
    bool operator()(const QStringList &row1, const QStringList &row2) const;
    enum { NumKeys = 3 };
    int keys[NumKeys];
    bool ascending[NumKeys];
};
```

SpreadsheetCompare클라스는 ()연산자를 실현하므로 특수한 클라스이다. 이것은 마치도 함수처럼 클라스를 사용하게 한다. 그러한 클라스들을 함수자(functor)라고 한다. 함수자들에 대하여 리해하기 위해 간단한 실례를 고찰한다.

```
class Square
{
public:
    int operator()(int x) const { return x * x; }
};
```

Square클라스는 하나의 함수 operator()(int)를 제공한다. 이 함수는 파라메터의 두제곱을 돌려준다. 이 함수를 compute(int)가 아니라 operator()(int)라고 명명함으로써 Square형 객체를 함수처럼 사용할수 있게 된다.

```
Square square;
int y = square(5);
```

그러면 SpreadsheetCompare를 포함하는 실례를 고찰하자.

```
QStringList row1, row2;
SpreadsheetCompare compare;
...
if (compare(row1, row2)) {
    // row1 is less than row2
}
```

compare객체는 일반적인 compare()함수처럼 사용된다. 또한 이것은 성원변수로서 보관하는 모든 정렬건과 정렬순서를 호출할수 있다.

다른 한가지 방법은 대역변수에 정렬건과 정렬순서를 보관하고 일반compare()함수를 리용하는것이다. 그러나 대역변수를 사용하는것은 그리 좋지 못하고 포착하기 어려운 오유를 일으킨다. 함수자는 stable_sort()와 같은 형판함수들을 리용하기 위한 더 강력한 수단이다.

여기에 표의 두 행을 비교하는데 사용하는 함수의 실현이 있다.

```
bool SpreadsheetCompare::operator() (const QStringList &row1, const QStringList &row2) const
{
   for (int i = 0; i < NumKeys; ++i) {
        int column = keys[i];
```

```
            if (column != -1) {
                if (row1[column] != row2[column]) {
                    if (ascending[i])
                        return row1[column] < row2[column];
                    else
                        return row1[column] > row2[column];
                }
            }
        }
        return false;
    }
```

첫 행이 둘째 행보다 작으면 true, 그렇지 않으면 false를 돌려준다. standardstable_sort()함수는 그 결과를 리용하여 정렬한다.

SpreadsheetCompare객체의 건과 자모순배렬은 MainWindow::sort()함수(2장에서 주었다.)에서 정의한다. 매개 건은 세포첨수 혹은 –1 ("None")을 보관한다.

매개 건에 대하여 2개의 행에 대응하는 세포항목들을 차례로 비교한다. 차이를 발견하면 true 혹은 false값을 돌려준다. 모든 비교가 같아지면 false를 돌려준다. stable_sort()함수는 비김(tie)상황을 해결하는데 정렬건의 순서를 리용하며 본래 row1이 row2을 앞서면 다른것보다 작은것으로서 아무것도 비교하지 않으며 결과에서도 여전히 row1이 row2를 선행한다. 이것은 std::stable_sort()이 std::sort()와 차이나는 점이다.

이리하여 Spreadsheet클라스를 완성하였다. 다음 절에서는 Cell클라스를 고찰한다. 이 클라스는 세포식들을 보관하는데 쓰이며 Spreadsheet가 세포의 식을 계산한 결과를 현시하기 위하여 호출하는 text()함수의 실현을 제공한다.

제6절. QTableItem의 파생클라스만들기

Cell클라스는 QTableItem을 계승한다. Cell클라스는 Spreadsheet와 잘 작업하도록 설계되지만 그 클라스에 대한 특별한 의존관계는 없고 리론적으로 QTable에서 사용할수 있다.

여기에 머리부파일이 있다.

```
#ifndef CELL_H
#define CELL_H
#include <qtable.h>
#include <qvariant.h>
class Cell : public QTableItem
{
public:
    Cell(QTable *table, const QString &formula);
```

```
    void setFormula(const QString &formula);
    QString formula() const;
    void setDirty();
    QString text() const;
    int alignment() const;
private:
    QVariant value() const;
    QVariant evalExpression(const QString &str, int &pos) const;
    QVariant evalTerm(const QString &str, int &pos) const;
    QVariant evalFactor(const QString &str, int &pos) const;
    QString formulaStr;
    mutable QVariant cachedValue;
    mutable bool cacheIsDirty;
};
#endif
```

Cell클라스는 3개의 비공개변수들을 추가하여 QTableItem을 확장한다.

· formulaStr는 세포의 식을 QString으로 보관한다.

· cachedValue는 세포의 값을 QVariant로서 보관한다.

· cacheIsDirty는 cachedValue가 갱신되지 않으면 true이다.

QVariant형은 C++와 Qt형의 값들을 보관할수 있다. 일부 세포들은 double값을 가지고 다른 일부 세포들은 QString값을 가지므로 이 형을 리용한다.

cachedValue와 cacheIsDirty변수를 C++ mutable예약어에 의해 선언하여 const함수들에서 변경할수 있게 한다. 또한 좀 비효과적이지만 text()를 호출할 때마다 그 값을 다시 계산할수 있다.

클라스선언에 Q_OBJECT마크로가 없다. Cell은 일반C++클라스로서 신호나 처리부가 없다. 사실상 QTableItem은 QObject를 계승하지 않으므로 Cell에는 신호와 처리부가 없다. Qt의 항목클라스들은 QObject를 계승하지 않음으로써 간접비를 최소로 유지한다. 신호와 처리부가 요구되면 항목들을 포함하는 창문부품으로 실현하거나 례외적으로 QObject와의 다중계승을 리용하여 실현할수 있다.

여기에 cell.cpp의 선두가 있다.

```
#include <qlineedit.h>
#include <qregexp.h>
#include "cell.h"
Cell::Cell(QTable *table, const QString &formula) : QTableItem(table, OnTyping)
{
```

```
        setFormula(formula);
}
```

구성자는 QTable의 지적자와 식을 받아들인다. 지적자는 QTableItem구성자에로 넘어가고 후에 QTableItem::table()에서 호출할수 있다. 기초클라스구성자의 둘째 인수 OnTyping은 사용자가 현재세포에서 입력을 시작할 때 편집기를 펼친다는것을 의미한다.

```
void Cell::setFormula(const QString &formula)
{
 formulaStr = formula;
 cacheIsDirty = true;
}
```

setFormula()함수는 세포의 식을 설정한다. 또한 cacheIsDirty기발을 true로 설정하는데 이것은 유효값이 돌아오기전에 cachedValue를 다시 계산해야 한다는것을 의미한다. 이것은 Cell구성자와 Spreadsheet::setFormula()로부터 호출된다.

```
QString Cell::formula() const
{
     return formulaStr;
}
```

formula()함수는 Spreadsheet::formula()로부터 호출된다.

```
void Cell::setDirty()
{
     cacheIsDirty = true;
}
```

setDirty()함수는 세포의 값을 다시 계산하기 위하여 호출한다. 이 함수는 단지 cacheIsDirty를 true로 설정한다. 재계산은 실제로 필요할 때까지 수행되지 않는다.

```
QString Cell::text() const
{
    if (value().isValid())
          return value().toString();
    else
          return "####";
}
```

text()함수는 QTableItem으로부터 재정의된다. 이 함수는 표에 현시하려는 본문을 돌려준다. 이것은 value()에 기초하여 세포의 값을 계산한다. 값이 무효이면(식이 틀려서) "####"를 돌려준다.

text()에서 사용한 value()함수는 QVariant를 돌려준다. QVariant는 double, QString과 같은 각

이한 형의 값들을 보관할수 있고 가변형을 다른 형으로 변환하는 함수들을 제공한다. 례를 들면 double값을 보관하는 가변형에 대한 toString()호출은 double의 문자렬표시를 생성한다. 기정구성자를 리용하여 구성한 QVariant는 《무효한》가변형이다.

```
int Cell::alignment() const
{
    if (value().type() == QVariant::String)
        return AlignLeft | AlignVCenter;
    else
        return AlignRight | AlignVCenter;
}
```

alignment()함수는 QTableItem로부터 재정의된다. 이 함수는 세포본문의 배렬방법을 돌려준다. 문자렬에 대해서는 왼쪽정렬을, 수값에 대해서는 오른쪽정렬 설정한다. 또한 모든 값들을 수직으로 중심에 배렬한다.

```
const QVariant Invalid;
QVariant Cell::value() const
{
    if (cacheIsDirty) {
        cacheIsDirty = false;
        if (formulaStr.startsWith("'")) {
            cachedValue = formulaStr.mid(1);
        } else if (formulaStr.startsWith("=")) {
            cachedValue = Invalid;
            QString expr = formulaStr.mid(1);
            expr.replace(" ", "");
            int pos = 0;
            cachedValue = evalExpression(expr, pos);
            if (pos < (int) expr.length())
                cachedValue = Invalid;
        } else {
            bool ok;
            double d = formulaStr.toDouble(&ok);
            if (ok)
                cachedValue = d;
            else
                cachedValue = formulaStr;
```

```
            }
        }
        return cachedValue;
    }
```

value()비공개함수는 세포의 값을 돌려준다. cacheIsDirty가 true이면 값을 다시 계산할 필요가 있다.

식이 =로 시작하면 문자렬을 위치 1로부터 취하고 그것이 포함하는 공백을 삭제한다. 그다음 evalExpression()을 호출하여 식의 값을 계산한다. pos인수는 참고로 넘어가고 이것은 문장해석을 시작하는 문자의 위치를 가리킨다. evalExpression()의 호출후에 pos는 성과적으로 문장해석된 식의 길이와 같다. 끝에 이르기전에 문장해석에서 실패하면 cachedValue를 Invalid로 설정한다.

식이 단일인용부호나 같이기호(=)로 시작되지 않으면 toDouble()에 의하여 류점수값으로 변환한다. 변환되면 cachedValue를 결과수값으로 설정하고 그렇지 않으면 cachedValue를 식문자렬로 설정한다. 례를 들면 식 1.50은 toDouble()에서 성공하여 1.5를 돌려주고 식 "World Population"은 실패하여 0.0을 돌려주게 한다.

value()함수는 const함수이다. cachedValue와 cacheIsValid를 mutable변수로 선언함으로써 const함수들에서 그 변수를 수정할수 있게 한다. value()를 비상수함수로 만들어 mutable예약어들을 제거할수 있으나 const함수 text()로부터 value()를 호출하므로 콤파일되지 않는다. 보통 C++에서 캐쉬과 mutable은 서로 협력한다.

이제는 식해석에 의존하지 않는 표계산프로그람을 만들었다. 이 절의 나머지 부분에서는 evalExpression()과 2개의 보조함수 evalTerm()과 evalFactor()를 설명한다. 코드는 좀 복잡하지만 이것을 포함하여 응용프로그람을 완성한다.

evalExpression()함수는 표계산식의 값을 돌려준다. 식은 +나 -연산자들로 구분된 하나이상의 항들로서 정의된다. 례를 들면 2*C5+D6은 첫 항이 2*C5이고 둘째 항이 D6인 식이다. 항자체는 *이나 /연산자들로 구분된 하나이상의 인수들로 정의된다. 례를 들면 2*C5는 첫 인수가 2, 둘째인수가 C5인 항이다. 끝으로 인수는 수(2), 세포위치(C5), 혹은 괄호안의 식(단항 – 부호가 앞에 있을수 있다)일수 있다. 식을 항들로, 항을 인수들로 분리하여 연산자들이 정확한 수속에 적용되도록 한다.

표계산식의 문법은 그림 4-12에서 정의된다. 문법의 매개 기호(Expression, Term 그리고 Factor)에 대하여 그것을 해석하는 대응하는 Cell성원함수가 있고 그 구조는 엄격히 문법을 따른다. 이런 방법으로 작성한 문장해석기는 재귀적인 하강식 문장해석기(recursive-descent parser)라고 부른다.

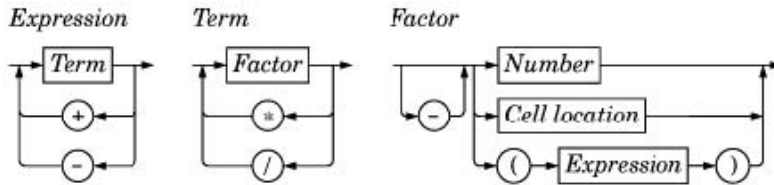



그림 4-12. 표계산식의 문법도표

식을 해석하는 함수 evalExpression()부터 고찰한다.

```
QVariant Cell::evalExpression(const QString &str, int &pos) const
{
    QVariant result = evalTerm(str, pos);
    while (pos < (int)str.length()) {
        QChar op = str[pos];
        if (op != '+' && op != '–') return result;
        ++pos;
        QVariant term = evalTerm(str, pos);
        if (result.type() == QVariant::Double && term.type() == QVariant::Double) {
          if (op == '+')
              result = result.toDouble() + term.toDouble();
          else
              result = result.toDouble()-term.toDouble();
        } else {
          result = Invalid;
        }
    }
    return result;
}
```

우선 evalTerm()을 호출하여 첫 항의 값을 얻는다. 다음 문자가 +나 -이면 두번째로 evalTerm()를 호출하고 그렇지 않으면 식은 하나의 항으로 이루어지고 그 값을 전체식의 값으로서 돌려준다. 처음 두개 항의 값을 얻은 후에 연산자에 따라 연산결과를 계산한다. 두 항이 double로 평가되면 결과를 double로 계산하고 그렇지 않으면 결과를 Invalid로 설정한다.

이 계산은 항이 더는 없을 때까지 계속된다. 이것은 더하기와 덜기가 왼쪽결합이므로 즉 1–2–3은 1–(2–3)이 아니라 (1–2)–3이므로 제대로 작업한다.

```
QVariant Cell::evalTerm(const QString &str, int &pos) const
{
    QVariant result = evalFactor(str, pos);
    while (pos < (int)str.length()) {
```

```
        QChar op = str[pos];
        if (op != '*' && op != '/') return result;
        ++pos;
        QVariant factor = evalFactor(str, pos);
        if (result.type() == QVariant::Double && factor.type() == QVariant::Double) {
          if (op == '*') {
               result = result.toDouble() * factor.toDouble();
          } else {
               if (factor.toDouble() == 0.0)
                    result = Invalid;
               else
                    result = result.toDouble() / factor.toDouble();
          }
       } else {
          result = Invalid;
       }
     }
     return result;
}
```

evalTerm()함수는 곱하기와 나누기를 취급하는것을 제외하면 evalExpression()과 아주 비슷하다. evalTerm()에서 포착하기 어려운 유일한 문제는 0에 의한 나누기를 피하는것이다. 류점수들의 등가성을 시험하는것은 반올림오유로 인하여 일반적으로 적당하지 않다. 령에 의한 나누기를 방지하는것이 안전하다.

```
QVariant Cell::evalFactor(const QString &str, int &pos) const
{
     QVariant result;
     bool negative = false;
     if (str[pos] == '–') {
          negative = true;
          ++pos;
     }
     if (str[pos] == '(') {
         ++pos;
         result = evalExpression(str, pos);
         if (str[pos] != ')')
```

```
            result = Invalid;
        ++pos;
    } else {
        QRegExp regExp("[A–Za–z] [1–9] [0–9] {0,2}");
        QString token;
        while (str[pos].isLetterOrNumber() || str[pos] == '.') {
            token += str[pos];
            ++pos;
        }
        if (regExp.exactMatch(token)) {
            int col = token[0].upper().unicode() – 'A';
            int row = token.mid(1).toInt() – 1;
            Cell *c = (Cell *)table()->item(row, col);
            if (c)
                result = c->value();
            else
                result = 0.0;
        } else {
            bool ok;
            result = token.toDouble(&ok);
            if (!ok)
                result = Invalid;
        }
    }
    if (negative) {
        if (result.type() == QVariant::Double)
            result = -result.toDouble();
        else
            result = Invalid;
    }
    return result;
}
```

evalFactor()함수는 evalExpression()과 evalTerm()보다 좀 더 복잡하다. 인수가 부수인가를 알리는것으로 시작한다. 그다음 열린괄호로 시작하는가 확인한다. 괄호로 시작하면 evalExpression()를 호출하여 괄호의 내용을 식으로 평가한다. 이것은 문장해석기에서 재귀가

발생하는 곳이며 evalExpression()은 evalTerm()를 호출하고 evalTerm()는 evalFactor()를, evalFactor()는 evalExpression()를 다시 호출한다.

인수가 겹쌓인 식이 아니면 세포위치나 수값일수 있는 다음 토큰을 꺼낸다. 토큰이 QRegExp와 일치하면 그것을 세포참고로 취하고 주어진 위치의 세포에 대하여 value()를 호출한다. 세포는 표의 어디에나 있을수 있으며 다른 세포들과 의존관계를 가질수 있다. 의존관계는 문제가 아니며 단순히 value()를 다시 호출하고 (dirty세포들에 대하여) 의존하는 모든 세포값들이 계산될 때까지 해석한다. 토큰이 세포위치가 아니면 그것을 수로 취급한다.

세포 A1이 식 =A1을 포함하면 어떤 일이 생기는가? 혹은 세포 A1이 =A2을, 세포 A2이 =A1을 포함한다면 어떤 일이 생기는가? 비록 순환하는 의존관계를 탐지하기 위한 특별한 코드를 쓰지 않았지만 문장해석기는 무효한 QVariant를 돌려줌으로써 이 경우를 훌륭히 처리한다. evalExpression()를 호출하기전에 value()에서 cacheIsDirty를 false로 설정하고 cachedValue를 Invalid로 설정하므로 이것은 제대로 작업한다. evalExpression()이 같은 세포에 대하여 재귀적으로 value()를 호출한다면 즉시 Invalid를 돌려주며 그때 전체 식은 Invalid로 평가된다.

이제는 식해석기를 완성하였다. sum()과 avg()와 같이 미리 정의된 표계산함수들을 조종하도록 확장하는것은 간단하다. 인수의 문법정의를 확장하면 된다. 다른 간단한 확장은 문자렬 연산수에 대하여 +연산자를 실현하는것이며 이것은 문법변경을 요구하지 않는다.

제5장. 사용자정의창문부품의 만들기

이 장에서는 Qt를 리용하여 사용자정의창문부품을 창조하는 방법을 설명한다. 사용자정의창문부품은 Qt의 현존창문부품들의 파생클라스를 만들거나 QWidget의 파생클라스를 직접 만들어서 창조할수 있다. 여기서는 두가지 수법을 모두 보여주며 사용자정의창문부품을 Qt Designer에 통합하여 Qt기본창문부품처럼 사용하는 방법을 보여준다. 끝으로 깜빡거림을 제거하는 강력한 기술 즉 2중완충기술을 사용하는 사용자정의창문부품을 제시한다.

제1절. Qt창문부품들의 사용자정의

일부 경우에 *Qt Designer*에서 Qt창문부품의 속성들을 설정하거나 함수들을 호출하여 될수록 전용화할것을 요구한다. 단순하면서 직접적인 해결은 관련한 창문부품클라스의 파생클라스를 만들고 요구에 맞게 적용하는것이다.

7F

그림 5-1. HexSpinBox창문부품

이 절에서는 16진수스핀칸을 개발하여 그 작업방법을 보여준다. QSpinBox는 오직 10진옹근수만 유지하지만 그 파생클라스를 만들어 16진수값들을 받아들이고 표시하는것은 아주 간단하다.

```
#ifndef HEXSPINBOX_H
#define HEXSPINBOX_H
#include <qspinbox.h>
class HexSpinBox : public QSpinBox
{
public:
    HexSpinBox(QWidget *parent, const char *name = 0);
protected:
    QString mapValueToText(int value);
    int mapTextToValue(bool *ok);
};
#endif
```

HexSpinBox는 QSpinBox로부터 대부분의 기능을 계승한다. 이 클라스는 전형적인 구성자를 제공하고 QSpinBox로부터 2개의 가상함수를 실현한다. 클라스가 자체의 신호와 처리부를 정의하지 않으므로 Q_OBJECT마크로가 필요없다.

```
#include <qvalidator.h>
#include "hexspinbox.h"
```

```
HexSpinBox::HexSpinBox(QWidget *parent, const char *name) : QSpinBox(parent, name)
{
    QRegExp regExp("[0-9A-Fa-f]+");
    setValidator(new QRegExpValidator(regExp, this));
    setRange(0, 255);
}
```

사용자는 스핀칸의 올리, 내리화살표를 찰칵하거나 스핀칸의 행편집기에 값을 입력하여 현재값을 수정할수 있다. 값을 직접 입력하는 경우 정확한 16진수를 입력하도록 제한하려고 한다. 그러기 위하여 범위 0～9, A～F, a～f로부터 하나이상의 문자들을 받아들이는 QRegExpValidator를 리용한다. 또한 기정범위를 0～255(0x00～0xFF)로 설정한다. 이것은 QSpinBox의 기정값 0～99보다 16진수 스핀칸에 더 적합하다.

```
QString HexSpinBox::mapValueToText(int value)
{
    return QString::number(value, 16).upper();
}
```

mapValueToText()함수는 옹근수값을 문자렬로 변환한다. QSpinBox는 이 함수를 호출하여 사용자가 스핀칸의 올리 혹은 내리화살표를 누를 때 스핀칸의 편집기부분을 갱신한다. 둘째 인수로서 16을 가지는 정적함수 QString::number()를 리용하여 소문자 16진수로 값을 변환하고 결과에 대하여 QString::upper()를 호출하여 그것을 대문자로 만든다.

```
int HexSpinBox::mapTextToValue(bool *ok)
{
    return text().toInt(ok, 16);
}
```

mapTextToValue()함수는 문자렬로부터 옹근수값에로의 역변환을 진행한다. 사용자가 스핀칸의 편집기부분에 값을 입력하고 Enter를 누르면 이 함수가 QSpinBox에 의해 호출된다. QString::toInt()함수에 의하여 현재본문(QSpinBox::text()로부터 돌아온다)을 16진옹근수값으로 변환한다.

변환이 성공하면 toInt()는 *ok를 true로 설정하고 그렇지 않으면 false로 설정한다.

이제는 16진스핀칸을 완료하였다. 다른 Qt창문부품만들기도 우와 같은 방법으로 진행한다. 즉 적당한 Qt창문부품을 선택하고 그 파생클라스를 만들고 가상함수들을 재정의하여 동작을 변경한다. 이 수법은 Qt프로그람작성에서 보편적이고 사실상 이미 4장에서 QTable의 파생클라스를 만들고 createEditor()와 endEdit()를 재정의할 때 사용하였다

제2절. QWidget의 파생클라스만들기

대부분의 사용자정의창문부품들은 그것이 Qt내부창문부품이든 HexSpinBox와 같은 사용자정의창문부품이든 순수 현존창문부품들의 결합이다. 현존창문부품들을 결합하여 건설하는 사

용자정의창문부품들은 보통 Qt Designer로 개발할수 있다. 즉

· Widget형판을 사용하여 새 폼을 작성한다.

· 폼에 필요한 창문부품들을 추가하고 배치한다.

· 신호와 처리부련결을 설정하고 .ui.h파일 혹은 파생클라스에 필요한 코드를 추가하여 요구되는 동작을 제공한다.

이것은 완전히 코드로도 수행할수 있다. 어떤 방법을 취하든간에 결과클라스는 QWidget로부터 직접 계승된다.

창문부품이 자체의 신호와 처리부를 가지지 않고 가상함수들을 재정의하지 않으면 파생클라스없이 현존창문부품들을 수집하여 창문부품을 간단히 조립할수도 있다. 그것은 1장에서 QHBox, QSpinBox, QSlider를 Age응용프로그람을 창조하는데 사용한 수법이다. 그러나 QHBox의 파생클라스를 간단히 만들고 파생클라스의 구성자에서 QSpinBox와 QSlider를 창조하였다.

Qt의 어떤 창문부품도 현재의 과제에 적합하지 않을 때와 현존창문부품들을 결합하거나 적용하여 요구되는 결과를 얻는 방법이 없을 때에는 요구되는 창문부품을 창조한다. 이것은 QWidget의 파생클라스를 만들고 몇가지 사건처리함수들을 재정의하여 창문부품을 그리고 마우스찰칵에 응답하게 하는 방법으로 달성한다. 이 수법은 완전히 자유롭게 창문부품의 표현과 동작을 정의하고 조종하게 한다. QLabel, QPushButton, QTable과 같은 Qt의 기본창문부품들은 이렇게 실현된다. Qt에 그것들이 존재하지 않으면 총체적으로는 가동환경에 의존하지 않는 방법으로 QWidget가 제공하는 공개함수들을 리용하여 자체로 창조할수 있다.

이러한 수법으로 사용자정의창문부품을 쓰는 방법을 보여주기 위하여 그림 5-2에 보여주는 IconEditor창문부품을 창조한다. IconEditor는 그림기호편집프로그람에서 사용할수 있는 창문부품이다.

그림 5-2. IconEditor창문부품

머리부파일부터 고찰하자.

```
#ifndef ICONEDITOR_H
#define ICONEDITOR_H
#include <qimage.h>
#include <qwidget.h>
class IconEditor : public QWidget
```

```
{
    Q_OBJECT
    Q_PROPERTY(QColor penColor READ penColor WRITE setPenColor)
    Q_PROPERTY(QImage iconImage READ iconImage WRITE setIconImage)
    Q_PROPERTY(int zoomFactor READ zoomFactor WRITE setZoomFactor)
public:
    IconEditor(QWidget *parent = 0, const char *name = 0);
    void setPenColor(const QColor &newColor);
    QColor penColor()const { return curColor; }
    void setZoomFactor(int newZoom);
    int zoomFactor() const { return zoom; }
    void setIconImage(const QImage &newImage);
    const QImage &iconImage() const { return image; }
    QSize sizeHint() const;
```

IconEditor클라스는 Q_PROPERTY()마크로를 리용하여 3개의 사용자정의속성 penColor, iconImage, zoomFactor를 선언한다. 매개 속성은 형, 《읽기》함수, 《쓰기》함수를 가진다. 례를 들면 penColor속성은 QColor이고 penColor()와 setPenColor()함수들에 의하여 읽고쓸수 있다.

Qt Designer에서 창문부품을 사용할 때 사용자정의속성들이 Qt Designer의 속성편집기에 QWidget로부터 계승되는 속성들아래에 나타난다. 속성은 QVariant가 유지하는 형일수 있다. Q_OBJECT마크로는 속성을 정의하는 클라스들에 필요하다.

```
protected:
    void mousePressEvent(QMouseEvent *event);
    void mouseMoveEvent(QMouseEvent *event);
    void paintEvent(QPaintEvent *event);
private:
    void drawImagePixel(QPainter *painter, int i, int j);
    void setImagePixel(const QPoint &pos, bool opaque);
    QColor curColor;
    QImage image;
    int zoom;
};
#endif
```

IconEditor는 QWidget로부터 3개의 보호함수를 재정의하고 여러개의 비공개함수와 변수들을 가진다. 3개의 비공개변수는 3가지 속성의 값들을 보관한다.

실현파일은 #include지령과 IconEditor의 구성자로 시작된다.

```
#include <qpainter.h>
#include "iconeditor.h"
IconEditor::IconEditor(QWidget *parent, const char *name)
      : QWidget(parent, name, WStaticContents)
{
      setSizePolicy(QSizePolicy::Minimum, QSizePolicy::Minimum);
      curColor = black;
      zoom = 8;
      image.create(16, 16, 32);
      image.fill(qRgba(0, 0, 0, 0));
      image.setAlphaBuffer(true);
}
```

구성자는 setSizePolicy()호출과 WStaticContents기발과 같은 미묘한 점들을 가진다. 그것들을 간단히 론의한다.

확대결수는 8로 설정되고 이것은 그림기호안의 매개 화소가 8×8인 바른4각형으로 표시된다는것을 의미한다. 펜색갈은 흑색으로 설정되고 흑색기호는 Qt클라스(QObject의 기초클라스)에서 미리 정의된 값이다.

그림기호자료는 image성원변수에 보관되고 setIconImage()와 iconImage()함수를 통하여 호출될수 있다. 일반적으로 그림기호편집프로그람은 그림기호파일을 열 때 setIconImage()를, 사용자가 그림기호파일을 닫으려고 할 때 그림기호를 얻기 위하여 iconImage()를 호출한다.

image변수는 QImage형이다. 그것을 16×16화소와 32bit깊이로 초기화하고 화상자료를 지우며 알파완충기를 허용한다.

QImage클라스는 하드웨어에 의존하지 않는 형식으로 화상을 보관한다. 이 클라스는 1bit, 8bit 혹은 32bit깊이를 사용하도록 설정할수 있다. 32bit깊이의 화상은 화소의 적, 록, 청요소에 각각 8bit를 리용한다. 나머지 8bit는 화소의 알파요소 즉 불투명도를 보관한다. 례를 들면 순수한 적색의 적, 록, 청과 알파요소는 값 255, 0, 0과 255를 가진다. Qt에서 이 색은

```
QRgb red = qRgba(255, 0, 0, 255);
```

과 같이 지정되거나

```
QRgb red = qRgb(255, 0, 0);
```

와 같이 지정될수 있다.

QRgb는 unsigned int의 형정의이고 qRgb()와 qRgba()는 32bit옹근수값으로 인수들을 결합하는 inline함수들이다. 또한 다음과 같이 쓸수도 있다.

```
QRgb red = 0xFFFF0000;
```

여기서 첫 FF는 알파요소에 대응하고 둘째 FF는 적색요소에 대응한다. IconEditor구성자에서는 알파요소로서 0을 리용하여 투명색으로 QImage를 채운다.

Qt는 색을 보관하는 2가지 형 즉 QRgb와 QColor를 제공한다. QRgb가 32bit화소자료를 보관하기 위해 QImage에서 사용된 형정의이지만 QColor는 사용가능한 수많은 함수들을 가지는 클라스로서 Qt에서 색을 보관하는데 널리 사용된다. IconEditor창문부품에서는 QImage를 론할 때 QRgb만 사용하며 penColor속성을 비롯하여 그밖의 모든것에는 QColor를 사용한다.

```
QSize IconEditor::sizeHint() const
{
    QSize size = zoom * image.size();
    if (zoom >= 3)
        size += QSize(1, 1);
    return size;
}
```

sizeHint()함수는 QWidget로부터 재정의되고 창문부품의 리상크기를 돌려준다. 여기서는 화상크기에 zoom결수를 곱하고 zoom결수가 3이상이면 각 방향으로 여유로서 1화소를 더하여 살창을 조절한다. (zoom결수가 2나 1이면 그림기호의 화소와 같은 칸으로 되므로 살창은 표시되지 않는다.)

대체로 창문부품의 크기암시는 배치체계와 결합되여 쓰일수 있다. Qt의 배치관리자들은 폼의 자식창문부품들을 배치할 때 창문부품의 크기암시를 최대한 고려하려고 한다. IconEditor가 좋은 배치도구로 되자면 그것이 신용할만한 크기암시를 알려주어야 한다.

크기암시와 함께 창문부품들을 늘이겠는가 줄이겠는가를 배치체계에 알리는 크기방략을 가진다. 구성자에서 수평과 수직크기방략으로서 QSizePolicy::Minimum을 리용하여 setSizePolicy()를 호출함으로써 창문부품의 크기암시가 실제로 그 최소크기이라는것을 배치관리자에 알려야 한다. 다시 말하여 창문부품은 필요하다면 그 크기를 늘일수 있지만 크기암시 아래로 줄일수 없다. 이것은 창문부품의 sizePolicy속성을 설정하여 Qt Designer에서 거절할수 있다. 여러가지 크기방략들의 의미는 6장(배치관리)에서 설명한다.

```
void IconEditor::setPenColor(const QColor &newColor)
{
    curColor = newColor;
}
```

setPenColor()함수는 현재펜색을 설정한다. 그 색은 새로 그리는 화소에 사용될수 있다.

```
void IconEditor::setIconImage(const QImage &newImage)
{
    if (newImage != image) {
        image = newImage.convertDepth(32);
        image.detach();
        update();
```

```
        updateGeometry();
    }
}
```

setIconImage()함수는 편집하려는 화상을 설정한다. 32bit화상이 아닌 경우에는 convertDepth()를 호출하여 32bit화상으로 만든다. 코드의 모든곳에서 화상자료가 32bit QRgb값으로 보관되는것으로 가정한다.

또한 detach()를 호출하여 화상에 보관된 자료의 깊이사본을 얻는다. 이것은 화상자료가 ROM에 보관될수 있기때문에 필요하다. QImage는 정확히 요구될 때만 화상을 복사하여 시간과 기억기를 절약하려고 한다. 이러한 최량화를 명시적공유라고 하며 11장의 4절에서 QMemArray<T>를 론의한다.

image변수를 설정한 후에 QWidget::update()를 호출하여 새 화상으로 창문부품을 다시 그리게 한다. 다음에 QWidget::updateGeometry()를 호출하여 크기암시가 달라진 창문부품을 포함한다는것을 배치관리자에 알린다. 그때 배치관리자는 자동적으로 새로운 크기암시를 받아들인다.

```
void IconEditor::setZoomFactor(int newZoom)
{
    if (newZoom < 1)
        newZoom = 1;
    if (newZoom != zoom) {
        zoom = newZoom;
        update();
        updateGeometry();
    }
}
```

setZoomFactor()함수는 화상의 zoom결수를 설정한다. 후에 0에 의한 나누기를 방지하기 위하여 1아래의 값은 수정한다. update()와 updateGeometry()를 호출하여 창문부품을 다시 그리고 크기암시변경에 대하여 배치관리자에 통지한다.

penColor(), iconImage(), zoomFactor()함수들은 머리부파일에서 inline함수들로 실현된다.

이제는 paintEvent()함수의 코드를 고찰한다. 이 함수는 IconEditor의 가장 중요한 함수로서 창문부품이 다시 그려질 때마다 호출된다. QWidget의 기정적으로 아무 일도 하지 않고 창문부품은 비여있다.

3장에서 언급한 contextMenuEvent()와 closeEvent()와 같이 paintEvent()는 사건처리함수이다. Qt는 다른 사건처리함수들을 수많이 가지고있다. 그 매개는 각이한 형의 사건에 대응한다.(7장에서는 사건처리에 대하여 깊이 설명한다.)

그리기사건이 발생하고 paintEvent()가 호출되는 경우는 다음과 같다.

· 창문부품이 처음으로 표시될 때 체계는 자동적으로 그리기사건을 생성하여 창문부품을 자체로 그리게 한다.

· 창문부품의 크기가 조절될 때 체계는 자동적으로 그리기사건을 생성한다.

· 창문부품이 다른 창문에 의하여 가리워졌다가 다시 나타나면(창문체계가 그 구역을 보관하지 않으면) 가리워졌던 구역에 대하여 그리기사건이 발생된다..

또한 QWidget::update()나 QWidget::repaint()를 호출하여 그리기사건을 강제로 발생시킬수도 있다. 이 두 함수들사이의 차이는 repaint()는 즉시 재그리기를 진행하지만 update()는 단순히 Qt가 다음에 사건들을 처리할 때 그리기사건을 발생한다는데 있다. (두 함수는 창문부품이 화면에 표시되지 않으면 아무 일도 하지 않는다.) update()가 여러번 호출되면 Qt는 깜빡거림을 피하기 위하여 그리기사건들을 하나의 그리기사건으로 압축한다. IconEditor에서는 늘 update()을 사용한다.

여기에 코드가 있다.

```
void IconEditor::paintEvent(QPaintEvent *)
{
    QPainter painter(this);
    if (zoom >= 3) {
        painter.setPen(colorGroup().foreground());
        for (int i = 0; i <= image.width(); ++i)
            painter.drawLine(zoom * i, 0, zoom * i, zoom * image.height());
        for (int j = 0; j <= image.height(); ++j)
            painter.drawLine(0, zoom * j, zoom * image.width(), zoom * j);
    }
    for (int i = 0; i < image.width(); ++i) {
        for (int j = 0; j < image.height(); ++j)
            drawImagePixel(&painter, i, j);
    }
}
```

창문부품에 QPainter객체를 창조한다. zoom결수가 3이상이면 QPainter::drawLine()함수에 의하여 살창을 구성하는 수평 및 수직선들을 그린다.

QPainter::drawLine()호출은 다음의 문법을 가지고있다.

painter.drawLine(*x1, y1, x2, y2*);

여기서 (*x1,y1*)는 직선의 한 끝의 위치, (*x2,y2*)는 다른 끝의 위치이다. 또한 4개의 int대신에 2개의 QPoint를 가지는 다중정의판의 함수가 있다.

Qt창문부품의 왼쪽웃구석 화소는 위치(0, 0)에 배치되고 오른쪽 아래구석의 화소는 (width()– 1, height()– 1)에 배치된다. 이것은 관례적인 데카르트자리표계와 비슷하지만 y축의 방

향이 바뀌고 프로그람작성에서 많이 사용된다. 이행, 비례, 회전, 자름과 같은 변환을 리용하여 QPainter의 자리표계를 변환할수도 있다. 이것은 8장(2차원 및 3차원도형처리)에서 설명한다.

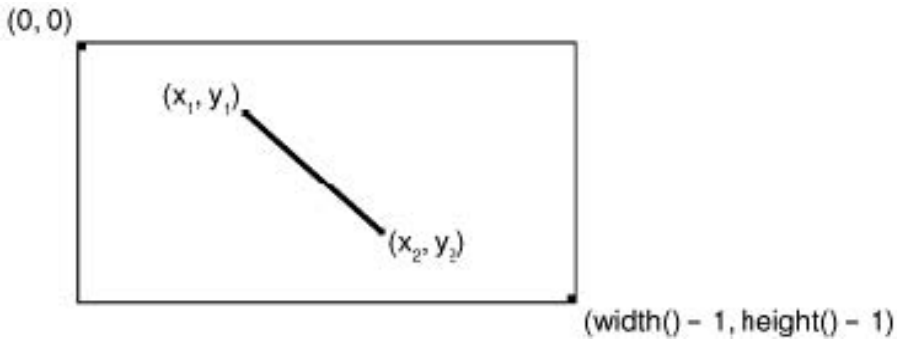



그림 5-3. QPainter에 의한 직선의 그리기

QPainter에 대하여 drawLine()를 호출하기전에 setPen()에 의하여 선의 색을 설정한다. 흑색이나 재색과 같은 색을 코드로 작성할수 있으나 창문부품의 조색판을 사용하는것이 더 좋은 수법이다.

매개 창문부품에는 사용할 색을 지정하는 조색판이 구비된다. 례를 들면 창문부품들의 배경색(보통 밝은재색)을 위한 조색판항목과 배경우에 표시되는 본문색(보통 흑색)을 위한 조색판항목이 있다. 기정으로 창문부품의 조색판은 창문체계의 색구조를 받아들인다. 조색판의 색을 리용하여 IconEditor가 사용자의 선택을 고려하도록 한다.

창문부품의 조색판은 3개의 색묶음 즉 능동, 비능동 및 금지로 이루어진다. 사용하려는 색묶음은 창문부품의 현재상태에 의존한다.

· 능동색묶음(active color group)은 현재 능동인 창문안에 있는 창문부품들에서 사용된다.

· 비능동색묶음(inactive color group)은 다른 창문들안의 창문부품들에서 사용된다.

· 금지색묶음(disable color group)은 임의의 창문의 금지된 창문부품들에서 사용된다.

QWidget::palette()함수는 창문부품의 조색판을 QPalette객체로서 돌려준다. 색묶음은 QPalette의 active(), inactive(), disabled()함수들을 통하여 유효로 되며 QColorGroup형이다. 편리상 QWidget::colorGroup()는 창문부품의 현재 상태에 대한 정확한 색묶음을 돌려주므로 간혹 조색판을 직접 호출해야 한다.

paintEvent()함수는 화상자체를 그리는것으로 끝난다. 이때 IconEditor::drawImagePixel()함수에 의하여 그림기호의 매개 화소를 색칠한 바른4각형을 그린다.

```
void IconEditor::drawImagePixel(QPainter *painter, int i, int j)
{
    QColor color;
    QRgb rgb = image.pixel(i, j);
    if (qAlpha(rgb) == 0)
        color = colorGroup().base();
```

```
        else
            color.setRgb(rgb);
        if (zoom >= 3) {
            painter->fillRect(zoom * i + 1, zoom * j + 1, zoom -1, zoom -1, color);
        } else {
            painter->fillRect(zoom * i, zoom * j, zoom, zoom, color);
        }
    }
```

drawImagePixel()함수는 QPainter에 의하여 확대된 화소를 그린다. i와 j파라메터는 창문부품이 아니라 QImage의 화소자리표이다. (zoom결수가 1이면 두개 자리표계는 정확히 일치한다.) 화소가 투명이면(알파요소가 0이면) 현재색묶음의 《기준》색(일반적으로 백색)으로 화소를 그리고 그렇지 않으면 화상안의 화소색을 리용한다. 그다음 QPainter::fillRect()을 호출하여 색으로 채운 바른4각형을 그린다. 살창이 표시되면 바른4각형을 량방향으로 1화소씩 감소하여 살창밖에 그리는것을 막는다.

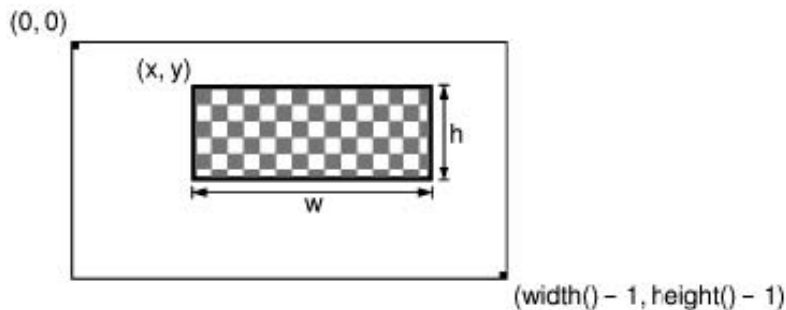



그림 5-4. QPainter에 의한 직4각형그리기

QPainter::fillRect()호출은 다음의 문법을 가진다.

painter->fillRect(*x, y, w, h, brush*);

여기서 (*x,y*)는 직4각형의 왼쪽웃구석의 위치, *w*×*h*는 직4각형의 크기이고 *brush*는 채우려는 색과 사용하려는 도색견본을 지정한다. 솔로서 QColor를 넘기여 솔리드도색견본을 얻는다.

```
    void IconEditor::mousePressEvent(QMouseEvent *event)
    {
        if (event->button() == LeftButton)
            setImagePixel(event->pos(), true);
        else if (event->button() == RightButton)
            setImagePixel(event->pos(), false);
    }
```

사용자가 마우스단추를 누를 때 체계는 《마우스누르기》사건을 생성한다. QWidget::mousePressEvent()를 재정의하여 사건에 응답할수 있으며 마우스유표밑의 화상화소를 설정하

거나 지울수 있다.

사용자가 왼쪽 마우스단추를 찰칵하면 둘째 인수를 true로 하여 비공개함수 setImagePixel()를 호출함으로써 화소를 현재펜색으로 설정한다. 사용자가 오른쪽 마우스단추를 찰칵하면 역시 setImagePixel()를 호출하지만 false를 넘기여 화소를 지운다.

```
void IconEditor::mouseMoveEvent(QMouseEvent *event)
{
    if (event->state() & LeftButton)
        setImagePixel(event->pos(), true);
    else if (event->state() & RightButton)
        setImagePixel(event->pos(), false);
}
```

mouseMoveEvent()는 《마우스이동》사건들을 처리한다. 기정으로 이 사건들은 사용자가 단추를 누르고있을 때에만 생성된다. QWidget::setMouseTracking()를 호출하여 이 동작을 변경할 수 있으나 이 실례에서는 그럴 필요가 없다.

왼쪽 혹은 오른쪽 마우스단추를 눌러서 화소를 설정하거나 지운다. 이때 단추를 누른 상태에서 화소를 끌고다니는 방법으로 화소를 설정하거나 지운다. 한번에 단추와 여러개의 건을 누를수 있으므로 QMouseEvent::state()가 돌려준 값은 마우스단추들(및 Shift와 Ctrl과 같은 수식건)의 비트별OR이다. &연산자를 리용하여 어느 단추를 눌렀는가 검사하고 단추를 눌렀으면 setImagePixel()를 호출한다.

```
void IconEditor::setImagePixel(const QPoint &pos, bool opaque)
{
    int i = pos.x() / zoom;
    int j = pos.y() / zoom;
    if (image.rect().contains(i, j)) {
        if (opaque)
            image.setPixel(i, j, penColor().rgb());
        else
            image.setPixel(i, j, qRgba(0, 0, 0, 0));
        QPainter painter(this); drawImagePixel(&painter, i, j);
    }
}
```

setImagePixel()함수는 mousePressEvent()와 mouseMoveEvent()로부터 호출되고 화소를 설정하거나 지운다. pos파라메터는 창문부품에서 마우스위치이다.

첫 단계는 마우스위치를 창문부품자리표로부터 화상자리표로 변환하는것이다. 이것은 마우스위치의 x와 y요소들을 zoom곁수로 나누어 수행한다. 다음에 점이 정확히 범위안에 있는

가 검사한다. 검사는 QImage::rect()와 QRect::contains()에 의해 쉽게 진행되는데 이때 i가 0과 image.width() – 1사이에, j는 0과 image.height() – 1사이에 있는가 검사한다.

opaque파라메터에 따라 화상의 화소를 설정하거나 지운다. 화소의 지우기는 실제로 화소를 투명으로 설정한다. 끝으로 drawImagePixel()를 호출하여 변경된 개별적인 화소를 다시 그린다.

성원함수들을 고찰하였으므로 구성자에서 사용한 WStaticContents기발에 대하여 보자. 이 기발은 Qt에 창문부품의 크기가 달라질 때 창문부품의 내용이 달라지지 않으며 그 내용은 창문부품의 왼쪽웃구석에 고착된다는것을 의미한다. Qt는 이 정보를 리용하여 창문부품크기가 달라질 때 이미 표시되여있는 구역을 필요없이 다시 그리는것을 피한다.

보통 창문부품크기가 변경될 때 Qt는 창문부품의 전체 보임구역에 대하여 그리기사건을 생성한다. 그러나 창문부품을 WStaticContents기발을 리용하여 창조하면 그리기사건의 령역(region)은 이전에 표시되지 않은 화소들로 제한된다. 창문부품의 크기가 더 작은 크기로 조절되면 그리기사건은 전혀 생성되지 않는다.

그림 5-5. WStaticContents창문부품의 크기조절

이제는 IconEditor창문부품이 완성되였다. 앞장들의 정보와 실례들을 사용하여 IconEditor를 자체의 창문으로, QMainWindow의 중심창문부품으로, 배치관리자안의 자식창문부품으로서, 혹은 QScrollView안의 자식창문부품으로서 사용하는 코드를 쓸수 있다. 다음 절에서는 이것을 Qt Designer에 통합하는 방법을 알게 된다.

제3절. 사용자정의창문부품을 Qt Designer와 통합하기

Qt Designer에서 사용자정의창문부품을 사용하려면 Qt Designer가 그것을 알고있어야 한다. 이것을 수행하는 수법에는 2가지 즉 《단순한 사용자정의창문부품》수법과 플라그인수법이 있다.

단순한 사용자정의창문부품수법은 Qt Designer에서 대화칸을 사용자정의창문부품에 대한 정보로 채우는것으로 시작한다. 그다음 창문부품은 Qt Designer를 리용하여 개발한 폼들에서 사용할수 있으나 그 창문부품은 폼을 편집하거나 미리보기하는동안 그림기호와 어두운 재색의 직4각형으로만 표시된다. 여기에 이 수법으로 HexSpinBox창문부품을 통합하는 방법이 있다.

① Tools|Custom|Edit Custom Widget를 찰칵하여 Qt Designer의 사용자정의창문부품편집기를 펼친다.

② NewWidget를 찰칵한다.

③ 클라스이름을 MyCustomWidget로부터 HexSpinBox로 변경하고 머리부파일을 mycustomwidget.h로부터 hexspinbox.h로 변경한다.

④ 크기암시를 (60, 20)으로 변경한다.

⑤ 크기방략을 (Minimum, Fixed)로 변경한다.

그다음 창문부품은 Qt Designer도구칸의 Custom Wigets부분에서 사용할수 있다.

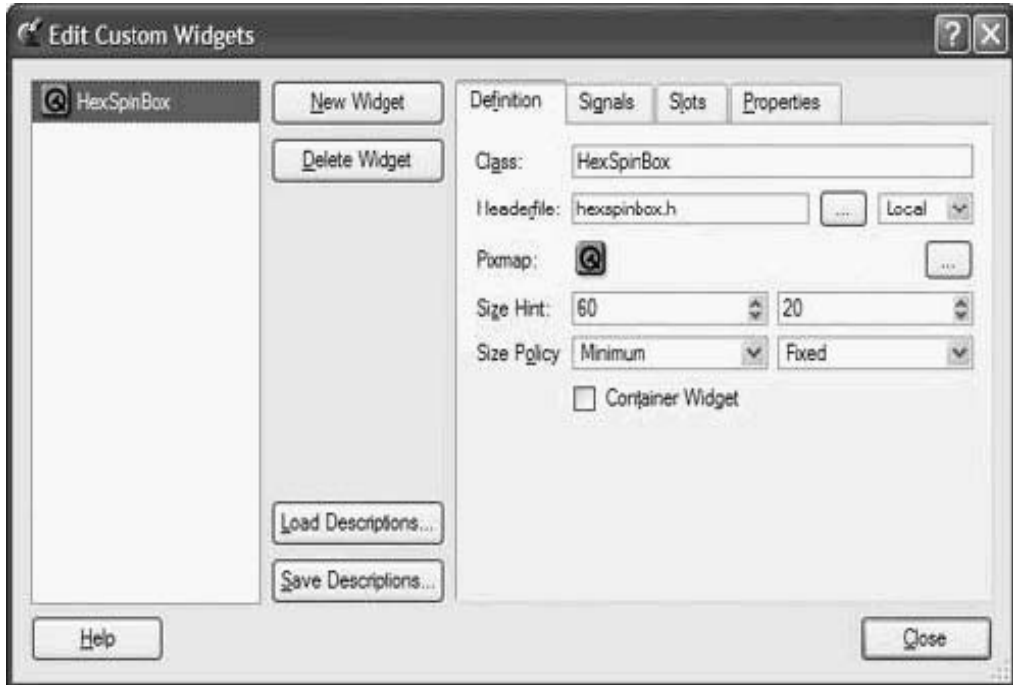



그림 5-6. Qt Designer의 사용자정의창문부품편집기

플라그인수법은 Qt Designer가 실행시에 적재하고 창문부품의 실례들을 창조하는데 사용할수 있는 플라그인서고의 창조를 요구한다. 실제창문부품은 그다음 폼을 편집하고 미리보기할 때 Qt Designer에 의해 사용된다. IconEditor을 플라그인으로서 통합하고 그 동작을 보여준다.

우선 QWidgetPlugin의 파생클라스를 만들고 일부 가상함수들을 재정의한다. 같은 원천파일에서 모든것을 수행한다. 플라그인원천코드는 iconeditorplugin이라는 등록부에 배치하고 IconEditor원천코드는 iconeditor라는 등록부에 배치한다.

여기에 머리부파일이 있다.

```
#include <qwidgetplugin.h>
#include "../iconeditor/iconeditor.h"
class IconEditorPlugin : public QWidgetPlugin
{
public:
    QStringList keys() const;
```

```
    QWidget *create(const QString &key, QWidget *parent, const char *name);
    QString includeFile(const QString &key) const;
    QString group(const QString &key) const;
    QIconSet iconSet(const QString &key) const;
    QString toolTip(const QString &key) const;
    QString whatsThis(const QString &key) const;
    bool isContainer(const QString &key) const;
};
```

IconEditorPlugin의 파생클라스는 IconEditor창문부품을 밀봉하는 factory클라스이다. 함수들은 Qt Designer에 의하여 클라스의 실례들을 창조하고 그에 대한 정보를 얻는데 사용된다.

```
QStringList IconEditorPlugin::keys() const
{
    return QStringList() << "IconEditor";
}
```

keys()함수는 플라그인에 의해 제공된 창문부품들의 목록을 돌려준다. 실례플라그인은 오직 IconEditor창문부품만 제공한다.

```
QWidget *IconEditorPlugin::create(const QString &, QWidget *parent, const char *name)
{
    return new IconEditor(parent, name);
}
```

create()함수는 Qt Designer에 의해 호출되고 창문부품클라스의 실례를 창조한다. 첫 인수는 창문부품의 클라스이름이다. 이 실례에서는 한개 클라스만 제공하므로 그것을 무시할수 있다. 다른 함수들도 첫 인수로서 클라스이름을 가진다.

```
QString IconEditorPlugin::includeFile(const QString &) const
{
    return "iconeditor.h";
}
```

includeFile()함수는 플라그인에 의해 밀봉된 지정된 창문부품의 머리부파일이름을 돌려준다. 머리부파일은 uic도구로 생성한 코드에 포함된다.

```
bool IconEditorPlugin::isContainer(const QString &) const
{
    return false;
}
```

isContainer()함수는 창문부품이 다른 창문부품들을 포함하면 true를 돌려주고 그렇지 않으면 false를 돌려준다. 례를 들면 QFrame은 다른 창문부품들을 포함하는 창문부품이다.

IconEditor가 다른 창문부품들을 포함할 필요가 없으므로 false를 돌려준다. 엄격히 말해서 창문부품은 다른 창문부품들을 포함할수 있으나 Qt Designer는 isContainer()가 false를 돌려줄 때 이것을 허용하지 않는다.

```
QString IconEditorPlugin::group(const QString &) const
{
    return "Plugin Widgets";
}
```

group()함수는 이 사용자정의창문부품이 속하는 도구칸그룹의 이름을 돌려준다. 이름을 이미 사용하고있으면 Qt Designer가 자동적으로 그 창문부품용의 새 그룹을 창조한다.

```
QIconSet IconEditorPlugin::iconSet(const QString &) const
{
    return QIconSet(QPixmap::fromMimeSource("iconeditor.png"));
}
```

iconSet()함수는 Qt Designer의 도구칸에서 사용자정의창문부품을 표시하는데 쓰이는 그림기호를 돌려준다.

```
QString IconEditorPlugin::toolTip(const QString &) const
{
    return "Icon Editor";
}
```

toolTip()함수는 마우스가 Qt Designer도구칸에서 사용자정의창문부품을 가리킬 때 표시하려는 도구암시를 돌려준다.

```
QString IconEditorPlugin::whatsThis(const QString &) const
{
    return "Widget for creating and editing icons";
}
```

whatsThis()함수는 Qt Designer가 현시하게 될 What's This?본문을 돌려준다.

```
    Q_EXPORT_PLUGIN(IconEditorPlugin)
```

플라그인클라스를 실현하는 원천파일의 끝에서 Q_EXPORT_PLUGIN()마크로에 의하여 플라그인을 Qt Designer에서 사용할수 있게 한다.

플라그인을 구축하기 위한 .pro파일은 다음과 같다.

```
TEMPLATE = lib
CONFIG += plugin
HEADERS = ../iconeditor/iconeditor.h
SOURCES = iconeditorplugin.cpp \
          ../iconeditor/iconeditor.cpp
```

```
IMAGES  =  images/iconeditor.png
DESTDIR = $(QTDIR)/plugins/designer
```

.pro파일은 QTDIR환경변수가 Qt를 설치한 등록부로 설정된것으로 가정한다. make 혹은 nmake에 의하여 플라그인을 구축할 때 Qt Designer의 plugins등록부에 자동적으로 그것을 설치한다.

플라그인을 구축하면 IconEditor창문부품을 Qt Designer에서 Qt의 기본창문부품들처럼 사용할수 있다.

제4절. 2중완충

2중완충(double buffering)은 활기있는 사용자대면부를 제공하고 깜빡거림을 제거하는데 사용할수 있는 기술이다. 깜빡거림은 아주 짧은 시간동안에 같은 화소를 여러번 각이한 색으로 그릴 때 발생한다. 깜빡거림이 오직 한 화소에서 발생하면 문제가 없지만 동시에 여러 화소들에서 발생하면 사용자를 혼란시킨다.

Qt가 그리기사건을 생성할 때 우선 창문부품을 조색판의 배경색으로 지운다. 그다음 창문부품은 paintEvent()에서 배경색과 같지 않은 화소만 그릴 필요가 있다. 이 2단계수법은 창문부품에서 오직 필요한 화소들만 그리고 다른 화소들을 고찰하지 않는다는것을 의미한다.

또한 2단계수법은 깜빡거림의 주요한 원천이다. 례를 들면 사용자가 창문부품크기를 조절하면 창문부품은 우선 그 자체를 모두 지우고 화소들을 그린다. 깜빡거림은 창문내용의 크기가 달라질 때 창문체계가 창문부품을 반복하여 지우고 그리므로 심해진다.

그림 5-7. 깜빡거림을 제거하지 않은 창문부품의 크기변경

IconEditor창문부품을 실현하는데 사용한 WStaticContents기발은 이 문제에 대한 하나의 해결책이지만 내용이 창문부품의 크기에의존하지 않는 창문부품들에만 쓰일수 있다. 그러한 창문부품은 드물다. 대부분의 창문부품들은 모든 유효공간을 차지하도록 자기 내용을 늘일수 있다. 또한 크기를 조절할 때 완전히 다시 그려야 한다. 깜빡거림을 피할수 있으나 해결책은 훨씬 더 복잡하다.

깜빡거림을 피하는 첫째 규칙은 WNoAutoErase기발을 리용하여 창문부품을 구성하는것이다. 이 기발은 Qt에 그리기사건에 앞서 창문부품을 지우지 말라고 알려준다. 그러면 낡은 화소가 변경되지 않은대로 남아있고 새로 나타난 화소들은 정의되지 않는다.

그림 5-8. WNoAutoErase창문부품의 크기변경

WNoAutoErase을 사용할 때 그리기처리함수가 모든 화소들을 명시적으로 설정하는것이 중요하다. 그리기사건에서 설정되지 않은 화소는 배경색을 요구하지 않는 이전의 값을 보유한다.

깜빡거림을 피하는 둘째 규칙은 모든 화소들을 한번만 그리는것이다. 이 요구를 실현하는 가장 간단한 방법은 화면밖의 픽스매프에 전체 창문부품을 그리고 그 픽스매프를 창문부품에 단번에 복사하는것이다. 이 수법을 리용하면 그리기가 화면밖에서 수행되므로 화소들이 여러번 그려져도 문제가 없다. 이것이 2중완충이다.

깜빡거림을 소거하기 위하여 사용자정의창문부품에 2중완충기능을 추가하는것은 간단하다. 원래의 그리기사건처리함수가 다음과 같다고 가정하자.

```
void MyWidget::paintEvent(QPaintEvent *)
{
    QPainter painter(this);
    drawMyStuff(&painter);
}
```

2중완충판은 다음과 같다.

```
void MyWidget::paintEvent(QPaintEvent *event)
{
    static QPixmap pixmap;
    QRect rect = event->rect();
    QSize newSize = rect.size().expandedTo(pixmap.size());
    pixmap.resize(newSize);
    pixmap.fill(this, rect.topLeft());
    QPainter painter (&pixmap, this);
    painter.translate(-rect.x(), -rect.y());
    drawMyStuff(&painter);
    bitBlt(this, rect.x(), rect.y(), &pixmap, 0, 0, rect.width(), rect.height());
}
```

우선 QPixmap의 크기를 변경하여 적어도 다시 그리려는 령역이나 경계직4각형의 크기로 만든다. (령역(region)은 흔히 직4각형이나 L형구역이지만 그것들의 임의의 결합일수 있다.)

QPixmap를 정적변수로 만들어 거기에 반복하여 기억기를 할당하고 해제하는것을 피한다. 같은 리유로 QPixmap를 절대로 줄이지 않으며 QSize::expandedTo()와 QPixmap::resize()의 호출은 그것이 늘 충분한 크기라는것을 담보한다. 크기변경후에 QPixmap::fill()에 의하여 QPixmap를 창문부품의 지우개색 혹은 배경픽스매프로 채운다. fill()의 둘째 인수는 QPixmap의 제일 웃준위화소가 창문부품의 어느 점에 대응하는가를 지정한다. (같은 색대신에 픽스매프를 리용하여 창문부품을 지운다면 차이나게 된다.)

QPixmap클라스는 QImage와 비슷하고 QWidget와도 비슷하다. QImage처럼 화상을 보관하지만 색깊이와 가능하다면 색변환을 가진 은페된 QWidget처럼 현시기에 배치한다. 창문체계가 8bit방식으로 실행되고있으면 모든 창문부품과 픽스매프들이 256색으로 제한되고 Qt는 자동적으로 24bit색명세를 8bit색들로 변환한다. (Qt의 색할당전략은 QApplication::setColorSpec()호출에 의해 조종된다.)

다음으로 QPainter를 창조하여 픽스매프에 조작한다. 이 지적자를 구성자에 넘기여 QPainter가 창문부품들의 서체와 같은 설정을 받아들이게 한다. 보통과 같이 QPainter를 리용하여 그리기전에 painter가 정확한 직4각형을 픽스매프에 그리도록 변환한다.

끝으로 bitBlt()대역함수에 의해 창문부품에 픽스매프를 복사한다. 이 함수의 이름은 비트블로크전송(bit-block transfer)으로부터 유래된것이다.

2중완충은 깜빡거림을 피하는데만 사용할수 있는것이 아니다. 창문부품의 그리기가 복잡하고 반복이 요구된다면 2중완충이 유익하다. 그때 창문부품을 리용하여 픽스매프를 디스크에 보관할수 있다. 즉 항상 다음의 그리기사건을 받을 준비를 하고있다가 그리기사건을 받을때마다 픽스매프를 창문부품에 복사한다. 이것은 전체 창문부품의 묘사를 반복하여 계산하지 않고 선택창을 그리는것과 같이 약간 수정하려고 하는 경우에 특별히 유익한 수법이다.

Plotter사용자정의창문부품을 고찰하는것으로 이 장을 끝낸다. 이 창문부품은 2중완충을 사용하며 건반사건처리, 수동배치, 자리표계와 같은 Qt프로그람작성의 다른 국면들도 보여준다.

Plotter창문부품은 자리표값들의 벡토르로서 지정된 하나이상의 곡선을 현시한다. 사용자는 화상우에 선택창을 그리고 Plotter는 선택창안에 들어있는 구역안에서 확대한다. 사용자는 그라프우의 한 점을 찰칵하고 마우스의 왼쪽단추를 누르면서 다른 위치로 끌고가서 마우스의 단추를 놓는 방법으로 선택창을 그린다.

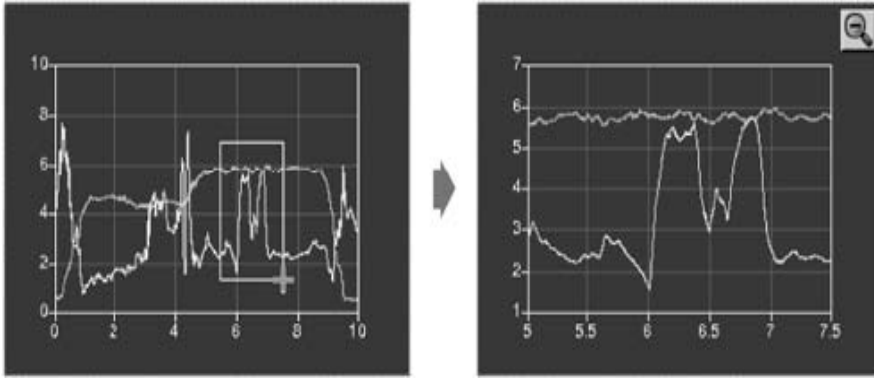



그림 5-9. Plotter창문부품에서 확대하기

사용자가 Zoom Out단추를 리용하여 축소하고 Zoom In단추에 의하여 다시 확대하여 선택창을 여러번 그리는 방법으로 반복하여 확대한다. Zoom In과 Zoom Out단추들은 그것들을 처음으로 사용하는 경우에 나타난다.

Plotter창문부품은 임의의 수의 곡선들에 대한 자료를 보관할수 있다. 또한 매개 요소가 특별한 확대준위에 대응하는 PlotSettings의 탄창을 보관한다.

plotter.h로부터 시작하여 클라스를 고찰하자.

```
#ifndef PLOTTER_H
#define PLOTTER_H
#include <qpixmap.h>
#include <qwidget.h>
#include <map>
#include <vector>
class QToolButton;
class PlotSettings;
typedef std::vector<double> CurveData;
```

표준<map>와 <vector>머리부파일을 포함한다. std이름공간의 모든 기호들을 대역이름공간에로 반입하지 않는다. 그것은 머리부파일에서 그렇게 하는것은 나쁜 형식이기때문이다.

CurveData를 std::vector<double>의 동의어로 정의한다. 곡선의 점들을 벡토르에 *x*와 *y*값들의 련속쌍으로 보관한다. 례를 들면 점(0, 24), (1, 44), (2,89)들에 의해 정의된 곡선은 벡토르 [0, 24, 1, 44, 2, 89]로 표시된다.

```
class Plotter : public QWidget
{
    Q_OBJECT
public:
    Plotter(QWidget *parent = 0, const char *name = 0, WFlags flags = 0);
    void setPlotSettings(const PlotSettings &settings);
```

```
    void setCurveData(int id, const CurveData &data);
    void clearCurve(int id);
    QSize minimumSizeHint() const;
    QSize sizeHint() const;
public slots:
    void zoomIn();
    void zoomOut();
```

도면(plot)을 설정하는 3개의 공개함수들과 확대와 축소를 위한 2개의 공개처리부를 제공한다. 또한 QWidget로부터 minimumSizeHint()와 sizeHint()를 재정의한다.

```
protected:
    void paintEvent(QPaintEvent *event);
    void resizeEvent(QResizeEvent *event);
    void mousePressEvent(QMouseEvent *event);
    void mouseMoveEvent(QMouseEvent *event);
    void mouseReleaseEvent(QMouseEvent *event);
    void keyPressEvent(QKeyEvent *event);
    void wheelEvent(QWheelEvent *event);
```

클라스의 보호부에서 재정의해야 할 QWidget사건처리함수들을 모두 선언한다.

```
private:
    void updateRubberBandRegion();
    void refreshPixmap();
    void drawGrid(QPainter *painter);
    void drawCurves(QPainter *painter);
    enum { Margin = 40 };
    QToolButton *zoomInButton;
    QToolButton *zoomOutButton;
    std::map<int, CurveData> curveMap;
    std::vector<PlotSettings> zoomStack;
    int curZoom;
    bool rubberBandIsShown;
    QRect rubberBandRect;
    QPixmap pixmap;
};
```

클라스의 비공개부에서 하나의 상수, 창문부품을 그리는 여러개의 함수들, 여러개의 성원변수들을 선언한다. Margin상수는 그라프주위에 공간을 제공하는데 쓰인다.

성원변수들중에는 QPixmap형의 pixmap가 있다. 이 변수는 화면에 표시되는것과 등가한 전체 창문부품의 그림의 사본을 보관한다. 도면은 늘 우선 화면밖의 픽스매프에 그려지고 그 다음 픽스매프가 창문부품에 복사된다.

```
class PlotSettings
{
public:
    PlotSettings();
    void scroll(int dx, int dy);
    void adjust();
    double spanX() const { return maxX -minX; }
    double spanY() const { return maxY -minY; }
    double minX;
    double maxX;
    int numXTicks;
    double minY;
    double maxY;
    int numYTicks;
private:
    void adjustAxis(double &min, double &max, int &numTicks);
};
#endif
```

PlotSettings클라스는 *x*와 *y*축의 범위와 이 축들의 눈금수를 지정한다. 그림 5-10은 PlotSettings객체와 Plotter창문부품상에서 눈금들사이의 대응관계를 보여준다.

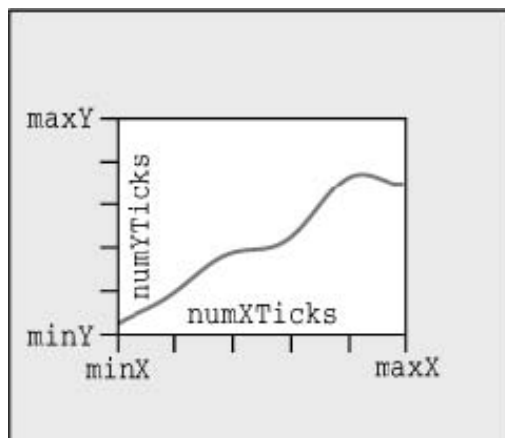



그림 5-10. PlotSettings의 성원변수

편리상 numXTicks와 numYTicks는 하나씩 차이난다. 즉 numXTicks이 5이면 Plotter는 실제

로 x축우에 6단위의 표식을 그린다. 이것은 후에 계산을 단순화한다.

그러면 실현파일을 고찰하자.

```
#include <qpainter.h>
#include <qstyle.h>
#include <qtoolbutton.h>
#include <cmath>
using namespace std;
#include "plotter.h"
```

기대한 머리부파일들을 포함하고 std이름공간의 기호들을 모두 대역이름공간에 반입한다.

```
Plotter::Plotter(QWidget *parent, const char *name, WFlags flags)
    : QWidget(parent, name, flags | WNoAutoErase)
{
    setBackgroundMode(PaletteDark);
    setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Expanding);
    setFocusPolicy(StrongFocus);
    rubberBandIsShown = false;
    zoomInButton = new QToolButton(this);
    zoomInButton->setIconSet(QPixmap::fromMimeSource("zoomin.png"));
    zoomInButton->adjustSize();
    connect(zoomInButton, SIGNAL(clicked()), this, SLOT(zoomIn()));
    zoomOutButton = new QToolButton(this);
    zoomOutButton->setIconSet( QPixmap::fromMimeSource("zoomout.png"));
    zoomOutButton->adjustSize();
    connect(zoomOutButton, SIGNAL(clicked()), this, SLOT(zoomOut()));
    setPlotSettings(PlotSetting());
}
```

Plotter는 parent와 name과 함께 flags파라메터를 가지고있다. 이 파라메터는 기초클라스구성자에 WNoAutoErase와 함께 넘긴다. 이 파라메터는 클라스의 사용자들이 창문틀과 제목띠를 구성할수 있게 하므로 독자적인 창문으로 쓰이는 창문부품들에 특히 쓸모있다.

setBackgroundMode()호출은 QWidget에게 창문부품을 지우기 위한 색으로서 조색판의 《어두운》요소를 《배경》요소대신에 사용하게 한다. 기초클라스 구성자에 WNoAutoErase기발을 넘기지만 Qt는 창문부품의 크기가 커질 때 paintEvent()가 새 화소들을 그릴 기회를 얻기 전에 새로 로출된 화소들을 채우는데 사용할수 있는 기정색을 여전히 요구한다. Plotter창문부품의 배경이 어두운 색이므로 이 화소들을 어두운 색으로 그리게 하는것이 상식이다.

setSizePolicy()호출은 창문부품의 크기방략을 량방향에서 QSizePolicy::Expanding로 설정한

다. 이것은 창문부품에 응답할수 있는 배치관리자에게 창문부품을 특별히 늘일 결심이지만 줄일수도 있다는것을 알린다. 이 설정은 많은 화면공간을 차지하는 창문부품들에 전형적이다. 기정값은 량방향에서 QSizePolicy::Preferred로서 창문부품이 크기암시의 크기로 선택되지만 최소크기암시까지 줄이거나 필요하다면 무한정 늘일수 있다는것을 의미한다.

setFocusPolicy()호출은 마우스를 찰칵하거나 Tab를 눌러서 창문부품이 초점을 받아들이게 한다. Plotter가 초점을 가질 때 건누르기사건들을 받아들인다. Plotter창문부품은 몇가지 건들을 인식한다. 즉 +는 확대, -는 축소, 화살건들은 올리, 내리, 왼쪽, 오른쪽 흘림.

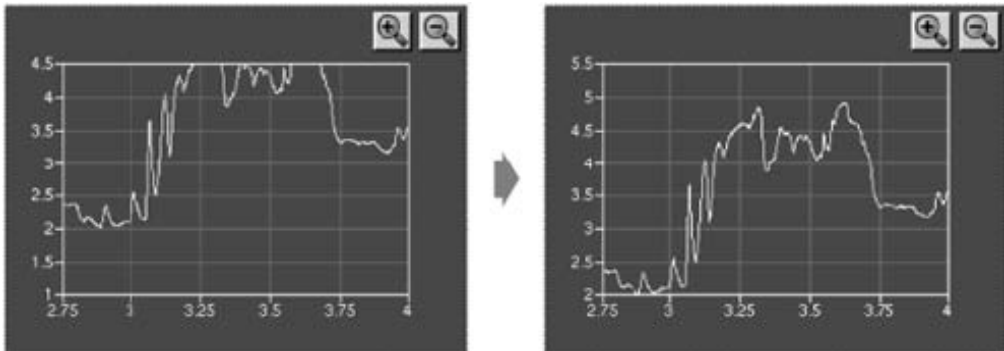



그림 5-11. Plotter창문부품의 흘림

여전히 구성자에서는 그림기호를 가지는 두개의 QToolButton을 창조한다. 이 단추들은 사용자가 zoomStack를 항행하게 한다. 단추의 그림기호들은 화상집합에 보관된다. Plotter창문부품을 사용하는 응용프로그람은 자기의 .pro파일에 항목을 요구한다.

```
IMAGES += images/zoomin.png \ images/zoomout.png
```

단추들에 대하여 adjustSize()를 호출하면 단추들의 크기를 크기암시의 크기로 설정한다.

마감에 setPlotSettings()의 호출은 초기화의 나머지 작업을 수행한다.

```
void Plotter::setPlotSettings(const PlotSettings &settings)
{
    zoomStack.resize(1);
    zoomStack[0] = settings;
    curZoom = 0;
    zoomInButton->hide();
    zoomOutButton->hide();
    refreshPixmap();
}
```

setPlotSettings()함수는 도면을 현시하는데 쓰이는 PlotSettings을 지정한다. 이 함수는 Plotter구성자에서 호출되고 클라스의 사용자들에 의해 호출될수 있다. Plotter는 기정확대준위에서 시작한다. 사용자가 확대할 때마다 새로운 PlotSettings실례가 창조되여 zoomStack에 넣어진다.

zoomStack는 2개의 성원변수들로 표시된다.

· zoomStack는 각이한 확대축소설정들을 vector<PlotSettings>로서 보관한다.

· curZoom은 zoomStack에서 현재 PlotSettings의 첨수를 보관한다.

setPlotSettings()호출후에 zoomStack는 오직 하나의 항목을 포함하고 Zoom In과 Zoom Out단추들은 숨겨진다. 이 단추들은 zoomIn()과 zoomOut()처리부들에서 그것들에 대하여 show()를 호출할 때까지 표시되지 않는다. (보통 제일 웃준위창문부품에 대하여 show()를 호출하여 자식들을 모두 표시하는것으로 충분하다. 그러나 자식창문부품에 대하여 명시적으로 hide()를 호출한 다음 show()를 호출할 때까지 그것은 은폐되여있다.)

refreshPixmap()호출은 현시를 갱신하는데 필요하다. 보통 update()를 호출하군 하지만 여기서는 언제나 최신식으로 QPixmap를 유지하려고 하므로 현저히 다른 일을 수행한다. 픽스매프를 다시 생성한 후에 refreshPixmap()는 update()를 호출하여 픽스매프를 창문부품에 보관한다.

```
void Plotter::zoomOut()
{
    if (curZoom > 0) {
        --curZoom;
        zoomOutButton->setEnabled(curZoom > 0);
        zoomInButton->setEnabled(true);
        zoomInButton->show();
        refreshPixmap();
    }
}
```

zoomOut()처리부는 그라프가 확대되면 축소한다. 이것은 현재의 확대준위를 감소시키고 그라프를 더 축소할수 있는가 없는가에 따라서 Zoom Out단추를 허용한다. Zoom In단추가 허용되여 표시되고 현시는 torefreshPixmap()호출과 함께 갱신된다.

```
void Plotter::zoomIn()
{
    if (curZoom > (int)zoomStack.size() -1) {
        ++curZoom;
        zoomInButton->setEnabled(curZoom < (int)zoomStack.size() -1);
        zoomOutButton->setEnabled(true);
        zoomOutButton->show();
        refreshPixmap();
    }
}
```

사용자가 이전에 확대하였다가 다시 축소하면 다음 확대준위를 위한 PlotSettings는

zoomStack에 있게 되고 확대할수 있다. (그렇지 않으면 선택창에 의하여 아직도 확대할수 있다.)

이 처리부는 curZoom을 증가시켜 zoomStack에로 1준위 더 깊이 들어가고 좀 더 확대할수 있는가에 따라서 Zoom In단추를 허용이나 금지로 설정하고 Zoom Out단추를 허용하고 표시한다. 다시 refreshPixmap()를 호출하여 Plotter가 최근의 확대설정을 사용할수 있게 한다.

```
void Plotter::setCurveData(int id, const CurveData &data)
{
    curveMap[id] = data;
    refreshPixmap();
}
```

setCurveData()함수는 주어진 ID의 곡선자료를 설정한다. 같은 ID를 가지는 곡선이 Plotter에 이미 존재한다면 그것은 새로운 곡선자료로 교체되며 그렇지 않으면 새 곡선은 그저 삽입된다. 곡선들은 map<int, CurveData>형의 curveMap성원변수에 보관된다.

다시 update()가 아니라 자체의 refreshPixmap()함수를 호출하여 현시를 갱신한다.

```
void Plotter::clearCurve(int id)
{
    curveMap.erase(id);
    refreshPixmap();
}
```

clearCurve()함수는 curveMap에서 하나의 곡선을 지운다.

```
QSize Plotter::minimumSizeHint() const
{
    return QSize(4 * Margin, 4 * Margin);
}
```

minimumSizeHint()함수는 sizeHint()와 비슷하며 sizeHint()가 창문부품의 리상크기를 지정하듯이 minimumSizeHint()는 창문부품의 리상적인 최소크기를 지정한다. 배치관리자는 결코 창문부품의 크기를 최소크기암시이하로 줄이지 않는다.

돌려주는 값은 160×160으로서 4개의 모든 변에 여백과 도면자체에 공간을 허용한다. 그 크기이하이면 도면은 사용하기에는 너무 작다.

```
QSize Plotter::sizeHint() const
{
    return QSize(8 * Margin, 6 * Margin);
}
```

sizeHint()에서는 여백에 비례하고 4:3의 가로세로비를 가지는 《리상》크기를 돌려준다.

이로서 Plotter의 공개함수들과 처리부들의 설명을 끝낸다. 이제는 보호사건처리함수들을

고찰한다.

```
void Plotter::paintEvent(QPaintEvent *event)
{
    QMemArray<QRect> rects = event->region().rects();
    for (int i = 0; i < (int)rects.size(); ++i)
        bitBlt(this, rects[i].topLeft(), &pixmap, rects[i]);
    QPainter painter(this);
    if (rubberBandIsShown) {
        painter.setPen(colorGroup().light());
        painter.drawRect(rubberBandRect.normalize());
    }
    if (hasFocus()) {
        style().drawPrimitive(QStyle::PE_FocusRect, &painter, rect(), colorGroup(),
                              QStyle::Style_FocusAtBorder, colorGroup().dark());
    }
}
```

보통 paintEvent()는 모든 그리기를 수행하는 함수이다. 그러나 여기서 모든 도면그리기는 refreshPixmap()에서 미리 수행되므로 픽스매프를 창문부품에 복사하여 전체 도면을 간단히 표시할수 있다.

QRegion::rect()호출은 다시 그리려는 령역을 정의하는 QRect들의 배렬을 돌려준다. bitBlt()를 사용하여 픽스매프에서 창문부품으로 매개의 직4각형구역을 복사한다. bitBlt()대역함수는 다음과 같은 문법을 가진다.

bitBlt(*dest*, *destPos*, *source*, *sourceRect*);

여기서 *source*는 원천창문부품 혹은 픽스매프, *sourceRect*는 복사되여야 할 원천안의 직4각형, *dest*는 목적창문부품 혹은 픽스매프, *destPos*는 목적지의 왼쪽웃구석위치이다.

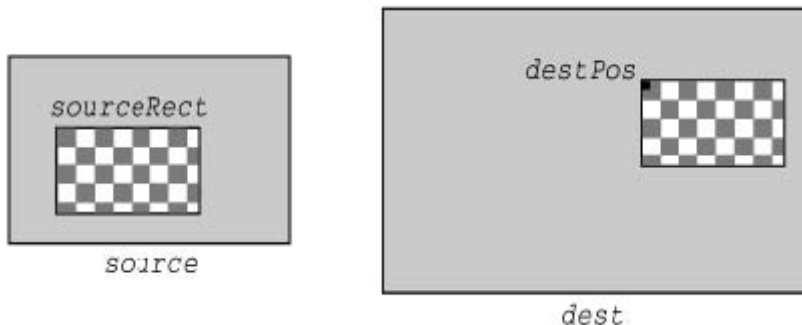



그림 5-12. 픽스매프와 창문부품들에 대한 임의의 직4각형들의 복사

앞의 코드부분에서 본것처럼 령역의 경계직4각형에 대하여 한번만 bitBlt()를 호출하도록 교정하였다. 그러나 마우스사건처리함수들안에서 update()를 호출하여 선택창을 반복하여 지우

고 다시 그리기하고 선택창의 테두리선이 기본적으로 4개의 작은 직4각형들(2개는 1화소폭의 직4각형, 두개는 1화소높이의 직4각형들)이므로 령역을 그것을 구성하는 직4각형들로 나누고 매개 직4각형에 대하여 bitBlt()를 호출하여 속도를 얻는다.

도면이 화면에 표시되면 선택창과 초점직4각형을 그 꼭대기에 그린다. 선택창에 대하여 창문부품의 현재색묶음으로부터 《가벼운》 요소를 펜색으로 사용하여 어두운 배경과 잘 대조되도록 한다. 화면밖의 픽스매프에 손을 대지 않고 직접 창문부품에 그린다. 초점직4각형은 첫 인수로서 PE_FocusRect를 가지는 창문부품형식의 drawPrimitive()함수를 리용하여 그린다.

QWidget::style()함수는 창문부품을 그리는데 사용할 창문부품형식을 돌려준다. Qt에서 창문부품형식은 QStyle의 파생클라스이다. 내부형식은 QWindowStyle, QWindowsXPStyle, QMotifStyle, QMacStyle을 포함한다. 매개 형식은 QStyle의 가상함수들을 재정의하여 그 형식이 모의하고있는 가동환경에 알맞는 방법으로 그리기를 수행한다. drawPrimitive()함수는 이러한 함수들중의 하나로서 조종판, 단추, 초점직4각형들과 같은 《원시요소》들을 그린다. 창문부품형식은 보통 한 응용프로그람의 모든 창문부품들에서 같지만(QApplication::style()) QWidget::setStyle()에 의하여 각 창문부품의 형식을 다시 설정할수 있다.

QStyle의 파생클라스를 만들어서 사용자정의형식을 정의할수 있다. 이것은 하나의 응용프로그람이나 한조의 응용프로그람들에 독특한 형식을 제공하도록 정의할수 있다. 일반적으로 목표가동환경의 원시형식을 사용할것을 권고하지만 Qt는 우수한 유연성을 제공한다.

Qt의 기본창문부품들은 거의 QStyle에만 기초하여 자체를 그린다. 이것은 기본창문부품들이 Qt에 의해 유지된 모든 가동환경들에서 원시적인 창문부품들처럼 보이기때문이다. 사용자정의창문부품들은 자체를 그리는데 QStyle을 사용하거나 기본Qt창문부품들을 자식창문부품들로서 리용함으로써 형식을 인식할수 있다. Plotter에서는 두 수법을 모두 사용한다. 즉 초점직4각형은 QStyle을 리용하여 그리며 Zoom In과 Zoom Out단추들은 기본Qt창문부품들이다.

```
void Plotter::resizeEvent(QResizeEvent *)
{
    int x = width() - (zoomInButton->width() + zoomOutButton->width() + 10);
    zoomInButton->move(x, 5);
    zoomOutButton->move(x + zoomInButton->width() + 5, 5);
    refreshPixmap();
}
```

Plotter창문부품의 크기가 변할 때 Qt는《크기변경》사건을 생성한다. 여기서는 resizeEvent()를 재정의하여 Plotter창문부품의 오른쪽 우에 Zoom In과 Zoom Out단추들을 배치한다.

Zoom In단추와 Zoom Out단추들을 옮기여 나란히 놓는다. 이때 5화소간격으로 갈라놓고 우에서 5화소 내려와서 부모창문부품의 오른변에 놓는다.

단추들을 자리표가 (0, 0)인 왼쪽웃구석에 놓으려고 하는 경우 Plotter구성자에서 단추들을 간단히 옮긴다. 그러나 오른쪽 웃구석에 유지하려고 하는 경우에 그 자리표들은 창문부품의

크기에 의존한다. 이리하여 resizeEvent()를 재정의하고 거기서 위치를 설정하는것이 필요하다.

Plotter구성자에서 단추들의 위치를 설정하지 않는다. 이것은 Qt가 항상 창문부품이 처음으로 표시되기전에 크기조절사건을 생성하므로 본질은 아니다.

resizeEvent()를 재정의하고 자식창문부품들을 수동으로 배치하는 다른 방법은 배치관리자(례를 들면 QGridLayout)를 사용하는것이다. 그러나 좀 더 복잡하고 더 많은 자원을 소비한다. 여기서 하는것처럼 창문부품들을 새로 작성할 때 자식창문부품들의 수동배치는 보통 옳은 수법이다.

끝으로 refreshPixmap()를 호출하여 픽스매프를 새로운 크기로 다시 그린다.

```
void Plotter::mousePressEvent(QMouseEvent *event)
{
    if (event->button() == LeftButton) {
        rubberBandIsShown = true;
        rubberBandRect.setTopLeft(event->pos());
        rubberBandRect.setBottomRight(event->pos());
        updateRubberBandRegion();
        setCursor(crossCursor);
    }
}
```

사용자가 왼쪽 마우스단추를 찰칵할 때 선택창의 현시를 시작한다. 이것은 rubberBandIsShown를 true로 설정하고 rubberBandRect성원변수를 현재마우스유표위치로 초기화하며 선택창을 그리도록 그리기사건을 발생시키고 마우스유표를 +형으로 변경하는 과정을 포함한다.

Qt는 마우스유표의 형태를 조종하는 2가지 기구를 제공한다.

· QWidget::setCursor()는 마우스가 특별한 창문부품의 우로 지날 때 사용하려는 유표형태를 설정한다. 창문부품에 유표가 설정되지 않으면 부모창문부품의 유표가 사용된다. 제일 웃준위 창문부품들의 기정유표는 화살유표이다.

· QApplication::setOverrideCursor()는 전체 응용프로그람용의 유표형태를 설정한다. 이때 restoreOverrideCursor()가 호출될 때까지 개별적인 창문부품들에 의하여 설정된 유표들은 무시된다.

4장에서는 waitCursor을 가지고 QApplication::setOverrideCursor()를 호출하여 응용프로그람의 유표를 표준기다림유표로 변경하였다.

```
void Plotter::mouseMoveEvent(QMouseEvent *event)
{
    if (event->state() & LeftButton) {
        updateRubberBandRegion();
```

```
            rubberBandRect.setBottomRight(event->pos());
            updateRubberBandRegion();
        }
    }
```

사용자가 왼쪽단추를 누른 상태에서 마우스유표를 옮길 때 updateRubberBandRegion()를 호출하여 그리기사건을 발생시켜 선택창이 있는 구역을 다시 그리고 마우스이동을 고려하여 rubberBandRect를 갱신하고 updateRubberBandRegion()를 두번째로 호출하여 선택창이 이동한 구역을 다시 그린다. 이것은 선택창을 효과적으로 지우고 그것을 새로운 자리표에 다시 그린다.

rubberBandRect변수는 QRect형이다. QRect는 다음과 같이 정의할수 있다.

(x,y,w,h)의 4요소 — 여기서 (x,y)는 직4각형의 왼쪽웃구석의 위치, $w \times h$는 그 크기 혹은 왼쪽웃구석과 오른쪽아래구석 자리표쌍.

여기서는 자리표쌍표시를 사용한다. 사용자가 처음에 찰칵한 점을 왼쪽웃구석으로 설정하고 현재의 마우스위치를 오른쪽아래구석으로 설정한다.

사용자가 마우스를 우로 혹은 왼쪽으로 옮기면 rubberBandRect의 명목상 오른쪽아래구석이 그 왼쪽웃구석의 우로 혹은 왼쪽으로 된다. 이러한 일이 생기면 QRect는 부수폭이나 높이를 가진다. QRect는 왼쪽웃구석과 오른쪽아래구석의 자리표들을 조절하여 부가 아닌 폭과 높이를 얻는 normalize()함수를 가지고있다.

```
    void Plotter::mouseReleaseEvent(QMouseEvent *event)
    {
        if (event->button() == LeftButton) {
            rubberBandIsShown = false;
            updateRubberBandRegion();
            unsetCursor();
            QRect rect = rubberBandRect.normalize();
            if (rect.width() < 4 || rect.height() < 4)
                return;
            rect.moveBy(-Margin, -Margin);
            PlotSettings prevSettings = zoomStack[curZoom];
            PlotSettings settings;
            double dx = prevSettings.spanX() / (width() -2 * Margin);
            double dy = prevSettings.spanY() / (height() -2 * Margin);
            settings.minX = prevSettings.minX + dx * rect.left();
            settings.maxX = prevSettings.minX + dx * rect.right();
            settings.minY = prevSettings.maxY -dy * rect.bottom();
```

```
            settings.maxY = prevSettings.maxY -dy * rect.top();
            settings.adjust();
            zoomStack.resize(curZoom + 1);
            zoomStack.push_back(settings);
            zoomIn();
        }
    }
```

사용자가 왼쪽마우스단추를 놓을 때 선택창을 지우고 표준화살유표를 되살린다. 선택창이 적어도 4×4이면 확대한다. 선택창이 그보다 작으면 사용자가 창문부품을 잘못 찰칵하거나 거기에 초점을 주는것과 같으므로 아무런 동작도 하지 않는다.

확대하는 코드는 좀 복잡하다. 이것은 동시에 두개의 자리표계 즉 창문부품자리표들과 Plotter자리표들을 취급하기때문이다. 여기서 수행하는 마우스의 작업은 rubberBandRect를 창문부품자리표로부터 Plotter자리표로 변환하는 일이다.

변환을 수행한 다음 PlotSettings::adjust()를 호출하여 수들을 순환하면서 매개 축에 대하여 의미를 가지는 걸음수를 찾는다.

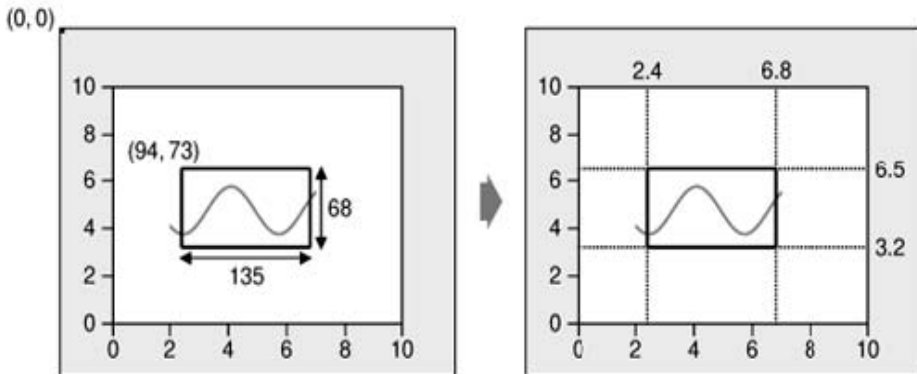



그림 5-13. 선택창을 창문부품으로부터 Plotter자리표로 변환

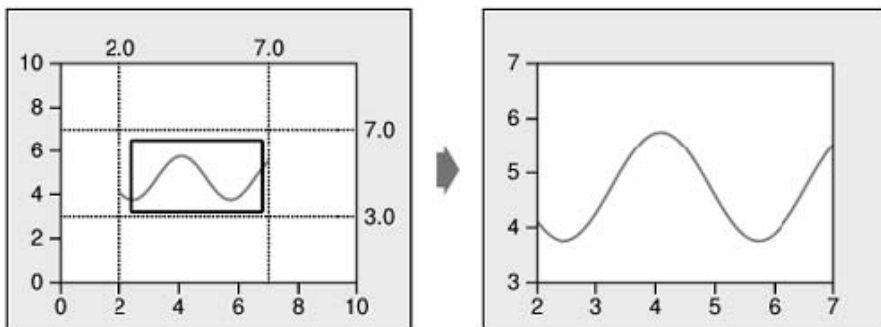



그림 5-14. Plotter자리표의 조절과 선택창에 대한 확대

그다음 확대한다. 확대는 우리가 방금 계산한 새로운 PlotSettings를 zoomStack의 꼭대기에 밀어넣고 zoomIn()을 호출하여 수행한다.

```
    void Plotter::keyPressEvent(QKeyEvent * event)
```

```
{
    switch (event->key()) {
        case Key_Plus:
            zoomIn();
            break;
        case Key_Minus:
            zoomOut();
            break;
        case Key_Left:
            zoomStack[curZoom].scroll(-1, 0);
            refreshPixmap();
            break;
        case Key_Right:
            zoomStack[curZoom].scroll(+1, 0);
            refreshPixmap();
            break;
        case Key_Down:
            zoomStack[curZoom].scroll(0, -1);
            refreshPixmap();
            break;
        case Key_Up:
            zoomStack[curZoom].scroll(0, +1);
            refreshPixmap();
            break;
        default: QWidget::keyPressEvent(event);
    }
}
```

사용자가 건을 눌러서 Plotter창문부품이 초점을 가질 때 keyPressEvent()함수가 호출된다. 여기서 이 함수를 재정의하여 6개건 즉 +, -, Up, Down, Left, Right에 응답한다. 조종되지 않는 건을 사용자가 누르면 기초클라스실현을 호출한다. 간단히 Shift, Ctrl, Alt수식건들을 무시하는데 이것들은 QKeyEvent::state()를 통하여 사용할수 있다.

```
void Plotter::wheelEvent(QWheelEvent *event)
{
    int numDegrees = event->delta() / 8;
    int numTicks = numDegrees / 15;
```

```
    if (event->orientation() == Horizontal)
        zoomStack[curZoom].scroll(numTicks, 0);
    else
        zoomStack[curZoom].scroll(0, numTicks);
    refreshPixmap();
}
```

바퀴사건은 마우스바퀴를 굴릴 때 발생한다. 대부분의 마우스는 오직 수직바퀴만 제공하지만 일부는 수평바퀴를 가지고있다. Qt는 두 종류의 바퀴를 유지한다. 바퀴사건들은 초점을 가지는 창문부품으로 간다. delta()함수는 바퀴가 회전한 거리를 8도단위로 돌려준다. 일반적으로 마우스는 15도걸음으로 작업한다.

바퀴마우스의 가장 일반적인 사용은 흘림띠를 이동시키는것이다. QScrollView(6장에서 설명)의 파생클라스를 만들어 흘림띠들을 제공할 때 QScrollView는 바퀴마우스사건들을 자동적으로 처리하므로 자체로 wheelEvent()를 재정의하지 않는다. 또한 QScrollView를 계승하는 QListView, QTable, QTextEdit와 같은 Qt클라스들은 추가적인 코드를 요구하지 않고도 바퀴사건들을 유지한다.

이것으로 사건처리함수들의 실현을 끝낸다. 이제는 비공개함수들을 고찰하자.

```
void Plotter::updateRubberBandRegion()
{
    QRect rect = rubberBandRect.normalize();
    update(rect.left(), rect.top(), rect.width(), 1);
    update(rect.left(), rect.top(), 1, rect.height());
    update(rect.left(), rect.bottom(), rect.width(), 1);
    update(rect.right(), rect.top(), 1, rect.height());
}
```

updateRubberBand()함수는 mousePressEvent(), mouseMoveEvent(), mouseReleaseEvent()로부터 호출되여 선택창을 지우거나 다시 그린다. 이것은 선택창에 의해 포함되는 작은 4개의 직4각형구역들에 대하여 그리기사건을 발생하는 update()의 4개의 호출로 이루어진다.

NOT를 리용한 선택창그리기

선택창을 그리는 일반적인 방법은 NOT (혹은 XOR)수학연산자를 사용하는것이다. 이 연산자는 선택창우의 매개 화소값을 반대의 bit견본으로 교체한다. 여기에 다음과 같은 일을 수행하는 updateRubberBandRegion()의 새로운 판이 있다.

```
void Plotter::updateRubberBandRegion()
{
    QPainter painter(this);
    painter.setRasterOp(NotROP);
```

```
        painter.drawRect(rubberBandRect.normalize());
    }
```

setRasterOp()호출은 painter의 라스터조작을 NotROP로 설정한다. 원래판에서는 기정값 CopyROP를 유지하였는데 이것은 QPainter에게 원시값에 새 값을 단순히 복사한다.

updateRubberBandRegion()를 같은 자리표에 대하여 두번 호출할 때 두개의 NOT가 서로 취소되므로 원래 화소들이 되살아난다.

NOT사용의 우점은 실현하기 쉽고 가리워진 구역들의 사본을 보관할 필요가 없다는것이다. 그러나 일반적으로 이것은 적용할수 없다. 례를 들면 선택창대신에 본문을 그린다면 본문은 아주 읽기 힘들다. 또한 NOT는 늘 좋은 결과를 생성하지 않는다. 례를 들면 중간재색은 그대로 남아있다. 끝으로 NOT는 Mac OS X에서 유지하지 않는다.

다른 수법은 선택창을 동화상의 접선으로 표시하는것이다. 이것은 흔히 화상조작프로그람들에서 사용한다. 그것은 화상에서 어떤 색이 발견되는가에 관계없이 좋은 대조를 제공하기때문이다. 이것을 Qt에서 수행하는 요령은 QObject::timerEvent()을 재정의하여 선택창을 지운 다음 그것을 다시 그리는데 있다.

```
void Plotter::refreshPixmap()
{
    pixmap.resize(size());
    pixmap.fill(this, 0, 0);
    QPainter painter(&pixmap, this);
    drawGrid(&painter);
    drawCurves(&painter);
    update();
}
```

refreshPixmap()함수는 화면밖의 픽스매프를 도면에 다시 그리고 현시를 갱신한다.

창문부품과 같은 크기를 가지도록 픽스매프의 크기를 조절하고 그것을 창문부품의 지우기색으로 채운다. 이 색은 Plotter구성자에서 setBackgroundMode()를 호출하므로 조색판의 《어두운》요소이다.

그다음 QPainter를 창조하여 픽스매프를 그리며 drawGrid()와 drawCurves()를 호출하여 그리기한다. 끝으로 update()를 호출하여 전체창문부품용으로 그리기사건을 발생한다. 픽스매프는 paintEvent(*)함수에서 창문부품에 복사된다.

```
void Plotter::drawGrid(QPainter *painter)
{
    QRect rect(Margin, Margin, width() -2 * Margin, height() -2 * Margin);
    PlotSettings settings = zoomStack[curZoom];
    QPen quiteDark = colorGroup().dark().light();
```

```
    QPen light = colorGroup().light();
    for (int i = 0; i <= settings.numXTicks; ++i) {
        int x = rect.left() + (i * (rect.width() -1) / settings.numXTicks);
        double label = settings.minX + (i * settings.spanX() / settings.numXTicks);
        painter->setPen(quiteDark);
        painter->drawLine(x, rect.top(), x, rect.bottom());
        painter->setPen(light);
        painter->drawLine(x, rect.bottom(), x, rect.bottom() + 5);
        painter->drawText(x -50, rect.bottom() + 5, 100, 15, AlignHCenter | AlignTop,
                          QString::number(label));
    }
    for (int j = 0; j <= settings.numYTicks; ++j) {
        int y = rect.bottom() -(j * (rect.height() -1) / settings.numYTicks);
        double label = settings.minY + (j * settings.spanY() / settings.numYTicks);
        painter->setPen(quiteDark);
        painter->drawLine(rect.left(), y, rect.right(), y);
        painter->setPen(light);
        painter->drawLine(rect.left() -5, y, rect.left(), y);
        painter->drawText(rect.left() -Margin, y -10, Margin -5, 20, AlignRight | AlignVCenter,
                          QString::number(label));
    }
    painter->drawRect(rect);
}
```

drawGrid()함수는 곡선과 축들의 배경에 살창을 그린다.

처음의 for순환은 살창의 수직선들과 *x*축의 눈금들을 그린다. 둘째for순환은 살창의 수평선들과 *y*축의 눈금들을 그린다. drawText()함수는 두 축에서 눈금표식에 대응하는 수값들을 그린다.

drawText()호출은 다음의 문법을 가진다.

painter.drawText(*x*, *y*, *w*, *h*, *alignment*, *text*);

여기서 (*x*, *y*, *w*, *h*)는 직4각형을 정의하고 *alignment*는 그 직4각형안에서 본문의 위치, *text*는 그리려는 본문이다.

```
void Plotter::drawCurves(QPainter *painter)
{
    static const QColor colorForIds[6] = { red, green, blue, cyan, magenta, yellow};
    PlotSettings settings = zoomStack[curZoom];
```

```
        QRect rect(Margin, Margin, width() -2 * Margin, height() -2 * Margin);
        painter->setClipRect(rect.x() + 1, rect.y() + 1, rect.width() -2, rect.height() -2);
        map<int, CurveData>::const_iterator it = curveMap.begin();
        while (it != curveMap.end()) {
            int id = (*it).first;
            const CurveData &data = (*it).second;
            int numPoints = 0;
            int maxPoints = data.size() / 2;
            QPointArray points(maxPoints);
            for (int i = 0; i < maxPoints; ++i) {
                double dx = data[2 * i] -settings.minX;
                double dy = data[2 * i + 1] -settings.minY;
                double x = rect.left() + (dx * (rect.width() -1) / settings.spanX());
                double y = rect.bottom() -(dy * (rect.height() -1) / settings.spanY());
                if (fabs(x) < 32768 && fabs(y) < 32768) {
                    points[numPoints] = QPoint((int)x, (int)y);
                    ++numPoints;
                }
            }
            points.truncate(numPoints);
            painter->setPen(colorForIds[(uint)id % 6]);
            painter->drawPolyline(points);
            ++it;
        }
    }
```

drawCurves()함수는 살창우에 곡선들을 그린다. setClipRect()호출로 시작하여 QPainter의 잘라내기령역을 곡선을 포함하는 직4각형(여백은 제외)으로 설정한다. 그때 QPainter는 구역밖에서 화소들에 대한 그리기조작을 무시한다.

다음에 곡선들을 모두 순환하면서 매개 곡선에 대하여 그것을 구성하는 (x,y)자리표쌍들을 순환한다. 반복자값의 첫 성원은 곡선의 ID를 제공하고 둘째 성원은 곡선자료를 제공한다.

for순환의 안쪽부분은 자리표쌍을 Plotter자리표로부터 창문부품자리표로 변환하여 합리적인 경계안에 놓여있는 points변수에 보관한다. 사용자가 많이 확대하면 16bit signed옹근수로 표시할수 없는 수들로 되며 이것은 창문체계에 따라서 부정확한 표시를 발생시킨다.

곡선의 모든 점들을 창문부품자리표로 변환하였다면 곡선의 펜색을 미리 정의된 색들중 하나로 설정하고 drawPolyline()를 호출하여 곡선의 모든 점들을 지나는 선을 그린다.

이것은 완성된 Plotter클라스이다. 남은것은 PlotSettings의 일부 함수들이다.

```
PlotSettings::PlotSettings()
{
    minX = 0.0;
    maxX = 10.0;
    numXTicks = 5;
    minY = 0.0;
    maxY = 10.0;
    numYTicks = 5;
}
```

PlotSettings구성자는 두개의 축을 범위가 0～10이고 5단위의 눈금표식을 가지도록 초기화한다.

```
void PlotSettings::scroll(int dx, int dy)
{
    double stepX = spanX() / numXTicks;
    minX += dx * stepX;
    maxX += dx * stepX;
    double stepY = spanY() / numYTicks;
    minY += dy * stepY;
    maxY += dy * stepY;
}
```

scroll()함수는 두 눈금사이의 간격에 주어진 수를 곱하여 minX, maxX, minY, maxY를 증가 혹은 감소시킨다. 이 함수는 Plotter::keyPressEvent()에서 흘림을 실현하는데 쓰인다.

```
void PlotSettings::adjust()
{
    adjustAxis(minX, maxX, numTicks);
    adjustAxis(minY, maxY, numYTicks);
}
```

adjust()함수는 mouseReleaseEvent()로부터 호출되여 minX, maxX, minY, maxY값들을 《좋은》값들로 둥그리기하고 매개 축에 적당한 눈금수를 결정한다. 비공개함수 adjustAxis()는 한번에 한개 축에 대하여 이 작업을 수행한다.

```
void PlotSettings::adjustAxis(double &min, double &max, int &numTicks)
{
    const int MinTicks = 4;
    double grossStep = (max -min) / MinTicks;
```

```
        double step = pow(10, floor(log10(grossStep)));
        if (5 * step < grossStep)
            step *= 5;
        else if (2 * step < grossStep)
            step *= 2;
        numTicks = (int) (ceil(max / step) -floor(min / step));
        min = floor(min / step) * step;
        max = ceil(max / step) * step;
    }
```

adjustAxis()함수는 min과 max파라메터들을 《좋은》 수들로 변환하고 numTicks파라메터를 주어진 [min,max]범위에 알맞도록 계산한 눈금수로 설정한다. adjustAxis()가 사본이 아니라 실제변수들(minX, maxX, numXTicks 등)을 수정해야 하므로 그 파라메터들은 비const참고이다.

adjustAxis()의 대부분의 코드는 단순히 2눈금(걸음)사이의 간격에 알맞는 값을 결정하려고 한다. 축을 따라 좋은 수들을 얻기 위해서는 세밀하게 걸음을 선택해야 한다. 례를 들면 걸음값 3.8은 축을 3.8의 배로 되게 한다. 10진수로 표식된 축들에서 좋은 걸음값들은 10^n, $2{\cdot}10^n$, 혹은 $5{\cdot}10^n$형식의 수이다.

걸음값에 대하여 최대값인 《총걸음》을 계산하는것으로 시작한다. 그다음 총걸음과 같거나 작은 10^n형식의 대응하는 수를 찾는다. 총걸음의 상용로그를 취하고 그 값을 그 수아래로 둥그리기한 수로 10을 제곱하여 이것을 수행한다. 례를 들면 총걸음이 236이면 log 236 = 2.37291...을 계산하고 그것을 둥그리여 2를 얻고 10^n형식의 후보걸음값으로서 $10^2 = 100$을 얻는다.

첫 후보걸음값이 있으면 그것을 리용하여 다른 2개 후보 2.10^n과 5.10^n을 얻는다. 우의 실례에서 2개의 다른 후보는 200과 500이다. 500은 총걸음보다 크므로 리용할수 없다. 그러나 200은 236보다 작으므로 이 실례에서는 걸음크기로 200을 사용한다.

걸음값으로부터 numTicks, min, max를 끌어내기는 아주 쉽다. 새로운 min값은 원래의 min을 걸음의 가장 가까운 배수쪽으로 아래로 둥그리기하여 얻으며 새로운 max값은 걸음의 가장 가까운 배수쪽으로 올리둥그리기하여 얻는다. 새로운 numTicks는 둥그리기한 min과 max값사이의 수이다. 례를 들면 min이 240이고 max가 1184이면 새 범위는 [200, 1200], 걸음표식은 5로 된다.

이 알고리듬은 일부 경우에 부분최량결과를 준다.

이 장에서는 현존Qt창문부품을 사용자정의하는 방법과 QWidget를 기초클라스로 사용하여 창문부품을 철저히 건설하는 방법을 설명하였다. 이미 2장에서 현존창문부품들로부터 창문부품을 구성하는 방법을 보았으며 6장에서 주제화상(theme)을 더 설명한다.

이 시점에서 우리는 Qt를 리용하여 GUI응용프로그람들을 쓰는 방법을 충분히 알고있다. 다음 장들부터는 Qt를 더 깊이 설명하여 Qt의 능력을 완전히 사용할수 있게 한다.

제6장. 배치관리

폼에 배치되는 매개 창문부품에는 적당한 크기와 위치가 주어져야 한다. 일부 큰 창문부품들은 사용자가 그 내용을 모두 호출하는데 흘림띠를 요구한다. 이 장에서는 폼에서 창문부품들의 각이한 배치방법을 고찰하며 류동가능창문과 MDI창문을 실현하는 방법을 설명한다.

제1절. 기본배치

Qt는 폼에 있는 자식창문부품들의 배치를 관리하는 3가지 기본방법 즉 절대위치지정, 수동배치 및 배치관리자를 제공한다. 그림 6-1에 실례로서 보여주는 Find File대화칸을 리용하여 이 수법들을 차례로 고찰한다.

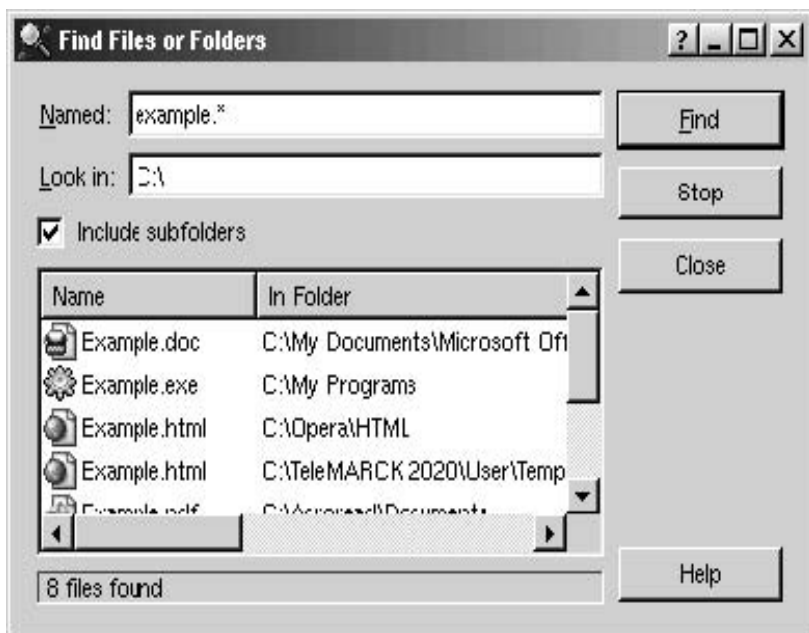



그림 6-1. Find File대화칸

절대위치지정은 창문부품들을 배치하는 가장 원시적인 방법이다. 절대위치지정은 폼의 자식창문부품들에 고정크기와 위치를 할당하고 폼에 고정크기를 할당하여 실현한다. 여기에는 FindFileDialog구성자에서 절대위치지정을 사용하는 방법을 보여준다.

```
FindFileDialog::FindFileDialog(QWidget *parent, const char *name) : QDialog(parent, name)
{
    ...
    namedLable->setGeometry(10, 10, 50, 20);
    namedLineEdit->setGeometry(70, 10, 200, 20);
    lookInLabel->setGeometry(10, 35, 50, 20);
    lookInLineEdit->setGeometry(70, 35, 200, 20);
```

```
    subfoldersCheckBox->setGeometry(10, 60, 260, 20);
    listView->setGeometry(10, 85, 260, 100);
    messageLabel->setGeometry(10, 190, 260, 20);
    findButton->setGeometry(275, 10, 80, 25);
    stopButton->setGeometry(275, 40, 80, 25);
    closeButton->setGeometry(275, 70, 80, 25);
    helpButton->setGeometry(275, 185, 80, 25);
    setFixedSize(365, 220);
}
```

절대위치지정에는 많은 결함이 있다. 첫째 문제는 사용자가 창문크기를 조절할수 없는것이다. 또 하나의 문제는 사용자가 큰 서체를 선택하거나 혹은 응용프로그람이 다른 언어로 번역되면 일부 본문이 잘리우는것이다. 그리고 이 수법은 또한 지루하게 위치와 크기를 계산하여야 한다.

절대위치를 지정하는 다른 수법은 수동배치이다. 수동배치(manual layout)에서는 창문부품들이 여전히 절대위치로 주어지지만 그 크기는 완전히 고정되지 않고 창문크기에 비례하여 정해진다. 이것은 폼의 resizeEvent()함수를 재정의하여 자식창문부품의 기하학적형태를 설정하는 방법으로 달성한다.

```
FindFileDialog::FindFileDialog(QWidget *parent, const char *name) : QDialog(parent, name)
{
    ...
    setMinimumSize(215, 170);
    resize(365, 220);
}
void FindFileDialog::resizeEvent(QResizeEvent *)
{
    int extraWidth = width() -minimumWidth();
    int extraHeight = height() -minimumHeight();
    namedLabel->setGeometry(10, 10, 50, 20);
    namedLineEdit->setGeometry(70, 10, 50 + extraWidth, 20);
    lookInLabel->setGeometry(10, 35, 50, 20);
    lookInLineEdit->setGeometry(70, 35, 50 + extraWidth, 20);
    subfoldersCheckBox->setGeometry(10, 60, 110 + extraWidth, 20);
    listView->setGeometry(10, 85, 110 + extraWidth, 50 + extraHeight);
    messageLabel->setGeometry(10, 140 + extraHeight, 110 + extraWidth, 20);
    findButton->setGeometry(125 + extraWidth, 10, 80, 25);
```

```
        stopButton->setGeometry(125 + extraWidth, 40, 80, 25);
        closeButton->setGeometry(125 + extraWidth, 70, 80, 25);
        helpButton->setGeometry(125 + extraWidth, 135 + extraHeight, 80, 25);
    }
```

FindFileDialog구성자에서는 폼의 최소크기를 215×170으로 설정하고 그 초기크기를 365×220으로 설정한다. resizeEvent()함수에서는 늘이려는 창문부품들에 여유공간을 준다.

절대위치지정처럼 수동배치는 프로그람작성자가 계산해야 할 고정정수들을 많이 요구한다. 이러한 코드를 쓰면 설계가 달라지는 경우에 아주 시끄럽다. 그리고 여전히 본문이 잘리울 위험이 있다. 그러한 위험은 자식창문부품들의 크기암시를 고려하여 피할수 있으나 그것은 코드를 더 복잡하게 만든다.

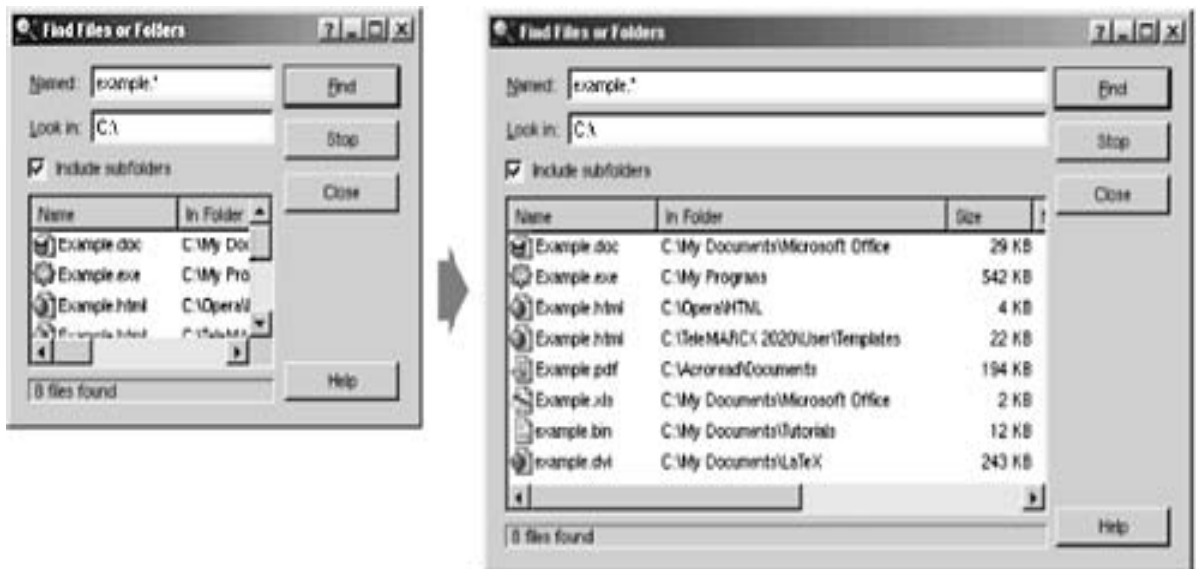



그림 6-2. 크기조절가능대화칸의 크기변경

폼우의 창문부품들을 배치하는 가장 좋은 해결책은 Qt의 배치관리자를 사용하는것이다. 배치관리자는 매개 형의 창문부품에 의식할수 있는 기정값들을 제공하며 창문부품의 서체, 형식, 내용에 의존하는 매개 창문부품의 크기암시를 고려한다. 또한 배치관리자는 최소크기와 최대크기를 고려하고 서체변경, 본문변경, 창문크기변경에 대하여 자동적으로 배치를 조절한다.

Qt는 3가지 배치관리자 즉 QHBoxLayout, QVBoxLayout, QGridLayout를 제공한다. 이 클라스들은 QLayout를 계승한다. QLayout는 배치의 기본틀거리를 제공한다. 3개의 클라스들은 모두 Qt Designer에 의해 완전히 유지되여있고 코드에서도 사용할수 있다. 2장에서 두 수법의 실례들을 제시하였다.

여기에 배치관리자들을 사용하는 FindFileDialog코드가 있다.

```
    FindFileDialog::FindFileDialog(QWidget *parent, const char *name) : QDialog(parent, name)
    {
        ...
```

```
    QGridLayout *leftLayout = new QGridLayout;
    leftLayout->addWidget(namedLabel, 0, 0);
    leftLayout->addWidget(namedLineEdit, 0, 1);
    leftLayout->addWidget(lookInLabel, 1, 0);
    leftLayout->addWidget(lookInLineEdit, 1, 1);
    leftLayout->addMultiCellWidget(subfoldersCheckBox, 2, 2, 0, 1);
    leftLayout->addMultiCellWidget(listView, 3, 3, 0, 1);
    leftLayout->addMultiCellWidget(messageLabel, 4, 4, 0, 1);
    QVBoxLayout *rightLayout = new QVBoxLayout;
    rightLayout->addWidget(findButton);
    rightLayout->addWidget(stopButton);
    rightLayout->addWidget(closeButton);
    rightLayout->addStretch(1);
    rightLayout->addWidget(helpButton);
    QHBoxLayout *mainLayout = new QHBoxLayout(this);
    mainLayout->setMargin(11);
    mainLayout->setSpacing(6);
    mainLayout->addLayout(leftLayout);
    mainLayout->addLayout(rightLayout);
}
```

배치는 QHBoxLayout, QGridLayout, QVBoxLayout에 의하여 조종된다. QGridLayout는 왼쪽에, QVBoxLayout는 오른쪽에 나란히 배치되고 QHBoxLayout는 바깥쪽에 배치된다. 대화칸주위의 여백은 11화소이고 자식창문부품들사이의 공백은 6화소이다.

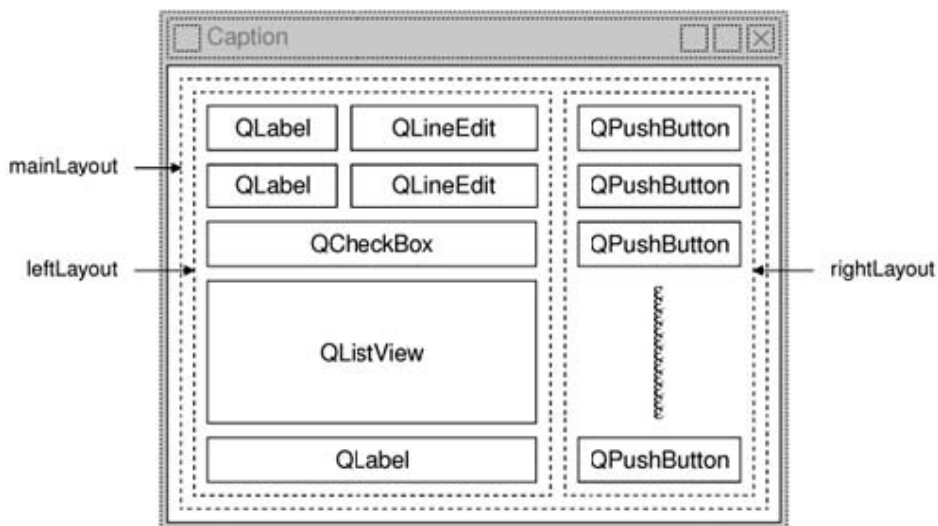



그림 6-3. Find File대화칸의 배치

QGridLayout는 2차원살창의 세포들에 대하여 작업한다. 배치관리자의 왼쪽웃구석에 있는 QLabel은 위치 (0, 0)에, 대응하는 QLineEdit는 위치 (0, 1)에 있다. QCheckBox는 두 칸에 전개되여 위치 (2, 0)과 (2, 1)의 세포들을 차지한다. 그아래에 QListView와 QLabel도 두 칸에 전개된다. addMultiCellWidget()호출은 다음의 문법을 가진다.

leftLayout->addMultiCellWidget(*widget*, *row1*, *row2*, *col1*, *col2*);

여기서 *widget*는 배치관리자에 삽입하려는 자식창문부품, (*row1, col1*)은 창문부품이 차지한 왼쪽웃구석세포, (*row2,col2*)는 창문부품이 차지한 오른쪽 아래구석세포이다.

Qt Designer에서 자식창문부품들을 적당한 위치에 배치하고 배치할것들을 선택한 다음 Layout|Lay Out Horizontally, Layout|Lay Out Vertically, 혹은 Layout|Lay Out in a Grid를 찰칵하여 대화칸을 시각적으로 창조할수 있다. 이 수법은 2장에서 표계산프로그람의 Go-to-Cell과 Sort 대화칸들을 창조하는데 사용되였다.

배치관리자의 사용은 우리가 지금까지 론의한것외에 다른 좋은 점도 있다. 배치관리자에 창문부품을 추가하거나 배치관리자에서 창문부품을 삭제할 때 배치관리자는 자동적으로 새로운 상황에 적응된다. 또한 자식창문부품에 대하여 hide()나 show()를 호출하여 적용할수도 있다. 자식창문부품의 크기암시가 달라지면 배치관리자들은 자동적으로 새 크기암시를 고려하여 재시행된다. 또한 배치관리자들은 폼의 자식창문부품들의 최소크기와 크기암시들에 기초하여 폼전체의 최소크기를 자동적으로 설정한다.

지금까지 제시한 매개의 실례에서는 배치관리자에 단순히 창문부품들을 넣고 수축자가 나머지 공간을 차지하도록 하였다. 흔히 우리가 바라는것과 꼭같이 배치할수는 없다. 그러한 상황에서는 배치하고있는 창문부품들의 크기방략과 크기암시들을 변경하여 배치를 조절할수 있다.

창문부품의 크기방략(size policy)은 배치체계에 창문부품을 늘이거나 줄이는 방법을 말해준다. Qt는 자기의 모든 기본창문부품들에 대하여 기정크기암시값들을 제공한다. 그러나 가능한 매개 배치관리자에 대하여 하나의 기정값만 고려할수 없으므로 폼우의 하나이상의 창문부품들에 대하여 크기방략들을 변경하는것은 개발자들에게 있어서 아직 관례로 되여있다. 크기암시는 수평 및 수직요소들을 모두 가진다. 매개 요소에 대한 가장 유효한 값들은 Fixed, Minimum, Maximum, Preferred, Expanding이다.

· Fixed는 창문부품을 늘이거나 줄일수 없다는것을 의미한다. 창문부품은 늘 그 크기암시의 크기로 된다.

· Minimum은 창문부품의 크기암시가 최소크기라는것을 의미한다. 창문부품은 크기암시아래로 줄일수 없지만 필요하다면 유효공간을 다 채우도록 늘일수 있다.

· Maximum은 창문부품의 크기암시가 최대크기라는것을 의미한다. 창문부품은 그 최소크기암시까지 줄일수 있다.

· Preferred는 창문부품의 크기암시가 적당한 크기라는것을 의미하지만 필요하다면 창문부

품을 늘이거나 줄일수 있다.

· Expanding은 창문부품을 늘이거나 줄일수 있는데 특히 늘일수 있다는것을 의미한다.

그림 6-4은 본문 "Some Text"를 표시하는 QLabel을 례들어 각이한 크기방략의 의미를 요약하여 보여준다.

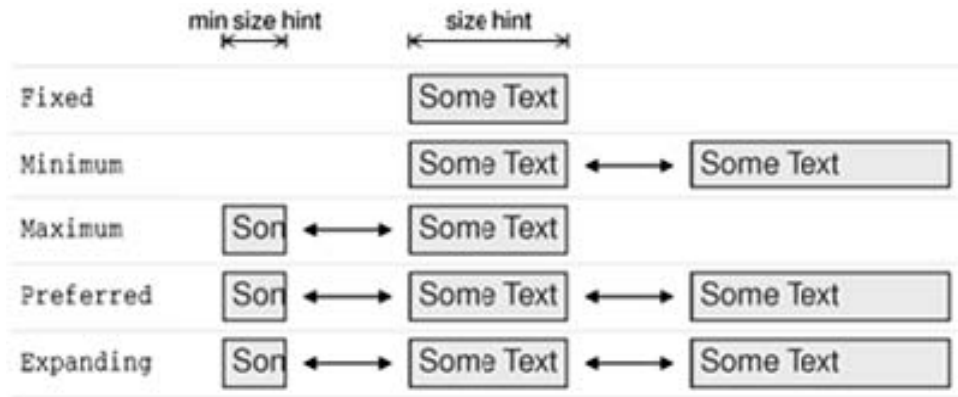



그림 6-4. 각이한 크기방략들의 의미

Preferred창문부품과 Expanding창문부품들을 모두 포함하는 폼의 크기를 조절할 때 Expanding창문부품들에 여유공간이 주어지고 Preferred창문부품들은 자기의 크기암시의 크기를 그대로 유지한다.

두가지 다른 크기방략 즉 MinimumExpanding과 Ignored가 있다. MinimumExpanding은 낡은 판의 Qt에서 아주 적은 경우에 필요하였지만 더는 사용하지 않으며 더 좋은 수법은 Expanding을 사용하여 minimumSizeHint()을 적당히 재정의하는것이다. Ignored는 Expanding과 비슷한데 창문부품의 크기암시를 무시한다는것이 다르다.

크기암시의 수평 및 수직요소들외에 QSizePolicy클라스는 수평 및 수직 늘임결수(stretch factor)를 둘다 보유한다. 이 늘임결수들은 폼을 확장할 때 각이한 자식창문부품들을 각이한 비률로 늘여야 한다는것을 지적하는데 쓰인다. 례를 들면 QTextEdit우에 QListView가 있고 QTextEdit를 QListView의 두배로 늘이려고 한다면 QTextEdit의 수직늘임결수를 2로, QListView의 수직늘임결수를 1로 설정한다.

배치에 영향을 주는 다른 수법은 자식창문부품들에 대하여 최소크기, 최대크기 또는 고정크기를 설정하는것이다. 배치관리자는 창문부품들을 배치할 때 이 제한들을 고려한다. 그리고 이것이 충분하지 않으면 늘 자식창문부품의 클라스로부터 파생하고 sizeHint()를 재정의하여 필요한 크기암시를 얻는다.

제2절. 분할기

분할기(splitter)는 다른 창문부품들을 포함하는 창문부품으로서 창문부품들을 분할기손잡이(handle)들에 의하여 분리한다. 사용자들은 손잡이를 끌기하여 분할기의 자식창문부품들의 크기를 변경할수 있다. 흔히 분할기들은 사용자에게 조종권을 더 주기 위한 배치관리자들의 또 하나의 수법으로 사용된다.

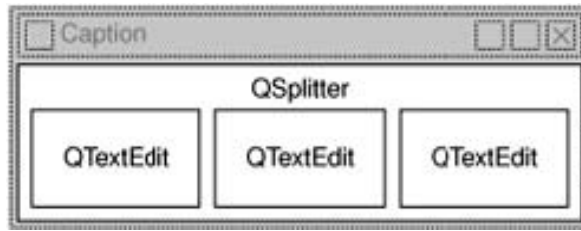



그림 6-5. Splitter응용프로그람의 창문부품들

Qt는 QSplitter창문부품에 의해 분할기들을 유지한다. QSplitter의 자식창문부품들은 그것이 창조되는 순서로 자동적으로 나란히(혹은 우아래로) 배치되고 이웃창문부품들사이에는 분할띠(splitter bar)들이 있다. 여기에 그림 6-5에 보여주는 창문을 창조하는 코드가 있다.

```
#include <qapplication.h>
#include <qsplitter.h>
#include <qtextedit.h>
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    QSplitter splitter(Qt::Horizontal);
    splitter.setCaption(QObject::tr("Splitter"));
    app.setMainWidget(&splitter);
    QTextEdit *firstEditor = new QTextEdit(&splitter);
    QTextEdit *secondEditor = new QTextEdit(&splitter);
    QTextEdit *thirdEditor = new QTextEdit(&splitter);
    splitter.show();
    return app.exec();
}
```

실례는 QSplitter창문부품에 의해 수평배치된 3개의 QTextEdit들로 이루어진다. 폼의 자식창문부품들을 배치만 하는 배치관리자와는 달리 QSplitter는 QWidget를 계승하고 다른 창문부품처럼 사용할수 있다.

QSplitter는 자식창문부품들을 수평 혹은 수직으로 배치한다. 복합배치는 수평 및 수직 QSplitter들을 겹쌓아서 얻을수 있다. 례를 들면 그림 6-6에 보여주는 Mail Client응용프로그람은 오른변에 수직QSplitter를 포함하는 수평QSplitter로 이루어진다.

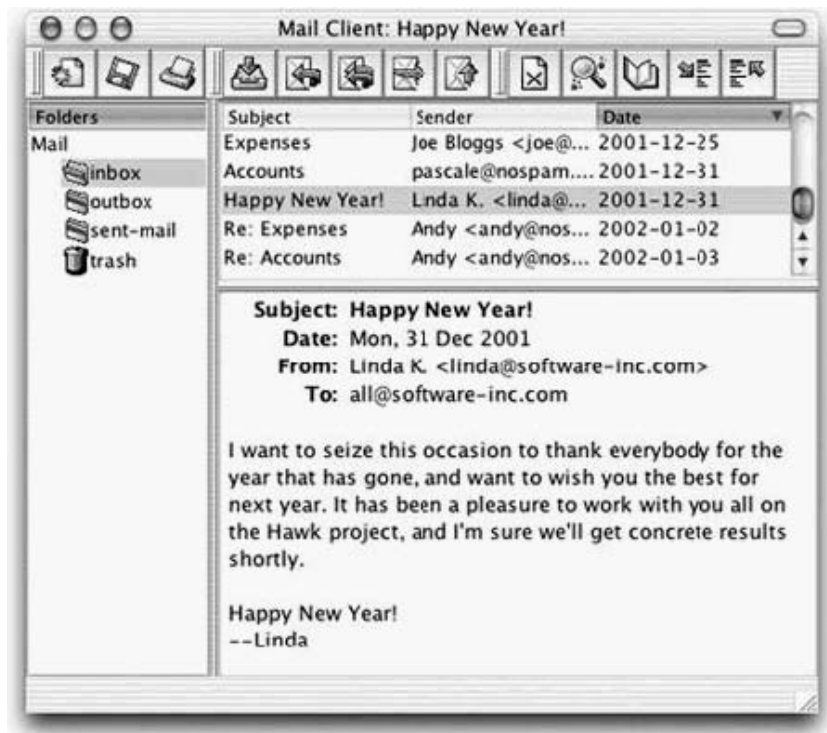



그림 6-6. Mac OS X에서 Mail Client응용프로그람

여기에 Mail Client응용프로그람에서 QMainWindow파생클라스구성자의 코드가 있다.

```
MailClient::MailClient(QWidget *parent, const char *name) : QMainWindow(parent, name)
{
    horizontalSplitter = new QSplitter(Horizontal, this);
    setCentralWidget(horizontalSplitter);
    foldersListView = new QListView(horizontalSplitter);
    foldersListView->addColumn(tr("Folders"));
    foldersListView->setResizeMode(QListView::AllColumns);
    verticalSplitter = new QSplitter(Vertical, horizontalSplitter);
    messagesListView = new QListView(verticalSplitter);
    messagesListView->addColumn(tr("Subject"));
    messagesListView->addColumn(tr("Sender"));
    messagesListView->addColumn(tr("Date"));
    messagesListView->setAllColumnsShowFocus(true);
    messagesListView->setShowSortIndicator(true);
    messagesListView->setResizeMode(QListView::AllColumns);
    textEdit = new QTextEdit(verticalSplitter);
    textEdit->setReadOnly(true);
```

```
        horizontalSplitter->setResizeMode(foldersListView, QSplitter::KeepSize);
        verticalSplitter->setResizeMode(messagesListView, QSplitter::KeepSize);
        ...
        readSettings();
    }
```

우선 수평QSplitter를 창조하고 그것을 QMainWindow의 중심창문부품으로 설정한다. 그다음 자식창문부품들과 그것들의 자식창문부품들을 창조한다.

사용자가 창문크기를 변경할 때 QSplitter는 보통 공간을 분배하여 관련된 자식창문부품들의 크기가 같아지게 한다. Mail Client실례에서는 이러한 동작을 요구하지 않고 그대신 자식창문부품들의 크기를 2개의 QListView가 관리할것을 요구하며 QTextEdit에 여유공간을 주려고 한다. 이것은 거의 마감에 2번의 setResizeMode()호출에 의해 달성된다.

응용프로그람이 기동할 때 QSplitter는 자식창문부품들에게 그것들의 초기크기에 기초하여 적당한 크기를 준다. QSplitter::setSizes()를 호출하여 분할기손잡이를 프로그람적으로 이동할수 있다. 또한 QSplitter클라스는 자기의 상태를 보관하고 응용프로그람이 다음번에 실행될 때 상태를 되살리는 수단을 제공한다. 여기에 Mail Client의 설정을 보관하는 writeSettings()함수가 있다.

```
    void MailClient::writeSettings()
    {
        QSettings settings;
        settings.setPath("software-inc.com", "MailClient");settings.beginGroup("/MailClient");
        QString str;
        QTextOStream out1(&str);
        out1 >> *horizontalSplitter;
        settings.writeEntry("/horizontalSplitter", str);
        QTextOStream out2(&str);
        out2 >> *verticalSplitter;
        settings.writeEntry("/verticalSplitter", str);
        settings.endGroup();
    }
```

여기에 대응하는 readSettings()함수가 있다.

```
    void MailClient::readSettings()
    {
        QSettings settings;
        settings.setPath("software-inc.com", "MailClient");settings.beginGroup("/MailClient");
        QString str1 = settings.readEntry("/horizontalSplitter");
```

```
    QTextIStream in1(&str1);
    in1 >> *horizontalSplitter;
    QString str2 = settings.readEntry("/verticalSplitter");
    QTextIStream in2(&str2);
    in2 >> *verticalSplitter;
    settings.endGroup();
}
```

이 함수들은 QTextIStream과 QTextOStream, 2개의 QTextStream편의파생클라스들에 기초하고있다.

기정으로 분할기손잡이는 사용자가 그것을 끌고다닐 때 선택창으로 표시되고 분할기손잡이의 어느 한쪽에 있는 창문부품의 크기는 사용자가 마우스단추를 놓을 때만 조절된다. 실제로 QSplitter가 자식창문부품들의 크기를 변경하기 위해서는 setOpaqueResize(true)를 호출하군한다.

QSplitter는 Qt Designer에 의하여 완전히 유지된다. 창문부품들을 분할기에 넣기 위해서는 요구되는 위치에 자식창문부품들을 대략적으로 배치하고 그것들을 선택한 다음 Layout|Lay Out Horizontally(in Splitter) 혹은 Layout|Lay Out Vertically(in Splitter)를 찰칵한다.

제3절. 창문부품탄창

배치를 관리하는 다른 하나의 창문부품은 QWidgetStack이다. 이 창문부품은 일련의 자식창문부품들이나 《페지》들을 포함하며 한번에 하나만 사용자에게 표시하고 다른 것은 숨긴다. 페지에는 0으로 시작하는 번호가 지정된다. 특정한 자식창문부품을 볼수 있게 하려면 페지번호나 자식창문부품의 지적자를 가지고 raiseWidget()를 호출한다.

QWidgetStack자체는 보이지 않고 사용자가 페지를 변경하기 위한 뚜렷한 시각적요소도 제공되는것이 없다. 그림 6-7의 작은 화살표들과 검은 회색틀은 QWidgetStack를 쉽게 설계하도록 하기 위하여 Qt Designer에서 제공한다.

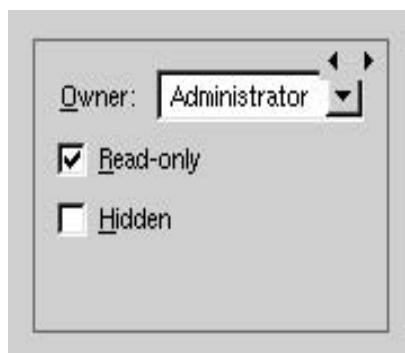



그림 6-7. QWidgetStack

그림 6-8에 보여주는 Configure대화칸은 QWidgetStack를 사용하는 실례이다. 대화칸의 왼쪽에는 QListBox, 오른쪽에는 QWidgetStack가 있다. QListBox의 매개 항목은 QWidgetStack안의

각이한 페지에 대응된다. 이와 같은 폼은 *Qt Designer*에서 생성하기 쉽다.

① Dialog 혹은 Widget형판에 기초하는 새로운 폼을 창조한다.

② 폼에 목록칸과 창문부품탄창을 각각 하나씩 추가한다.

③ 각 창문부품탄창페지를 자식창문부품들과 배치관리자들로 채운다. (새 페지를 창조하려면 오른쪽단추를 찰칵하고 Add Page를 선택한다. 페지들을 절환하려면 창문부품탄창의 오른쪽 웃끝에 배치된 아주 작은 왼쪽화살표 혹은 오른쪽화살표를 찰칵한다.)

④ 창문부품들을 수평배치한다.

⑤ 목록칸의 highlighted(int)신호를 창문부품탄창의 raiseWidget(int)처리부에 련결한다.

⑥ 목록칸의 currentItem속성값을 0으로 설정한다.

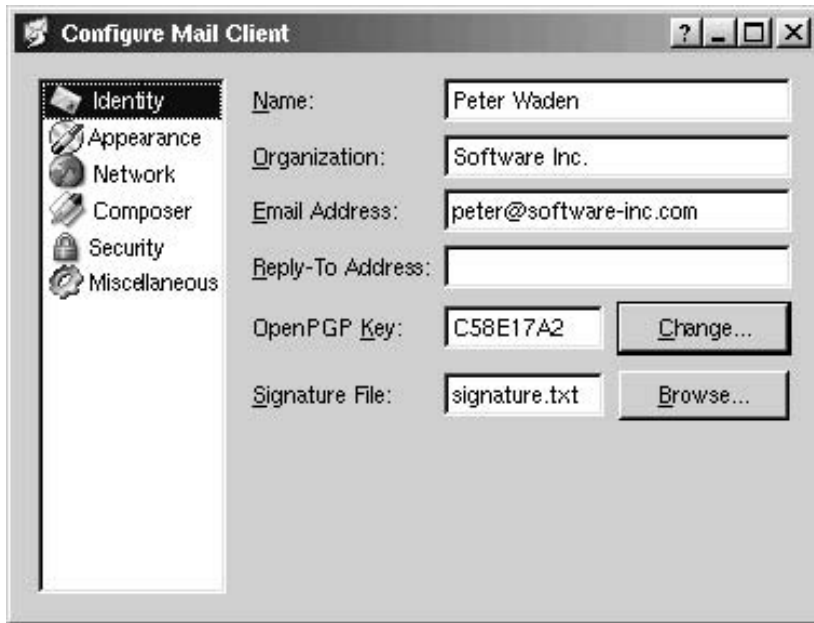



그림 6-8. Configure대화칸

미리 정의된 신호와 처리부에 의하여 페지절환을 실현하였으므로 대화칸은 Qt Designer에서 미리보기했을 때 정확한 페지절환동작을 보여준다.

제4절. 흘림보기

QScrollView클라스는 흘림가능한 보기구역, 2개의 흘림띠, 하나의 《구석》창문부품(보통 빈 QWidget)을 제공한다. 창문부품에 흘림띠들을 추가하려면 자체로 QScrollBar들의 실례를 만들고 흘림기능을 실현하기보다 QScrollView를 사용하는것이 훨씬 더 간단하다.

QScrollView를 사용하는 가장 간단한 방법은 흘림띠를 추가하려는 창문부품에 대하여 addChild()를 호출하는것이다. QScrollView은 자동적으로 창문부품을 보기구역(QScrollView::viewport()를 통하여 호출할수 있다)의 자식으로(이미 자식으로 설정되지 않는 경우에만) 설정한다. 실례로 5장에서 개발한 IconEditor창문부품주위에 흘림띠들이 필요하다면 다음과 같이 쓸수 있다.

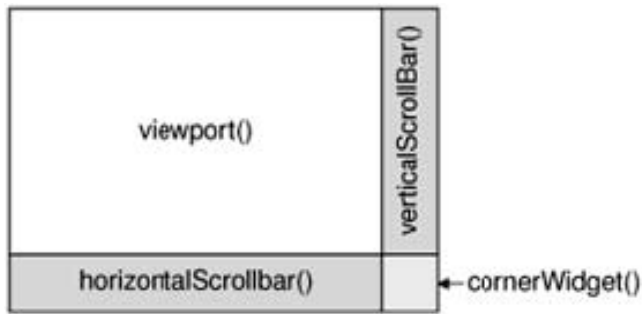



그림 6-9. QScrollView의 구성창문부품들

```
#include <qapplication.h>
#include <qscrollview.h>
#include "iconeditor.h"
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    QScrollView scrollView;
    scrollView.setCaption(QObject::tr("Icon Editor"));
    app.setMainWidget(&scrollView);
    IconEditor *iconEditor = new IconEditor;
    scrollView.addChild(iconEditor);
    scrollView.show();
    return app.exec();
}
```

기정으로 흘림띠들은 보기구역이 자식창문부품보다 작을 때에만 표시된다. 다음과 같이 코드를 써서 흘림띠들이 늘 표시되게 할수 있다.

```
    scrollView.setHScrollBarMode(QScrollView::AlwaysOn);
    scrollView.setVScrollBarMode(QScrollView::AlwaysOn);
```

자식창문부품의 크기암시가 달라질 때 QScrollView는 자동적으로 새로운 크기암시를 받아들인다.

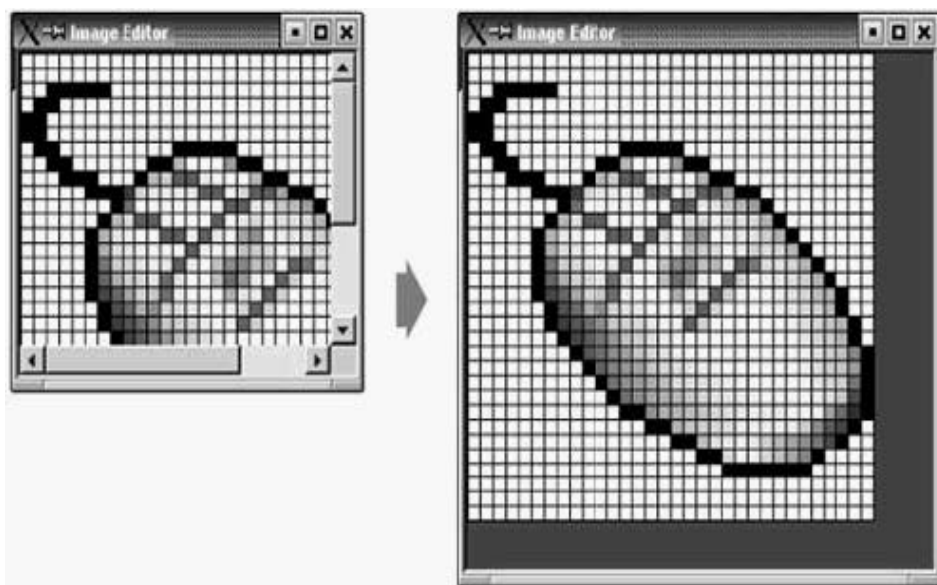



그림 6-10. QScrollView의 크기변경

창문부품을 가지는 QScrollView를 사용하는 다른 방법은 창문부품이 QScrollView를 계승하게 하고 내용을 그리는 drawContents()를 재정의하는것이다. 이것은 QIconView, QListBox, QListView, QTable, QTextEdit와 같은 Qt클라스들에서 사용한 수법이다. 일반적으로 창문부품이 흘림띠들을 요구한다면 QScrollView의 파생클라스를 실현하는것이 좋은 생각이다.

그 작업방법을 보여주기 위하여 새로운 판의 IconEditor클라스를 QScrollView파생클라스로서 실현한다. 새 클라스를 ImageEditor라고 부르고 그 흘림띠들이 큰 화상을 조종할수 있게 만든다.

```
#ifndef IMAGEEDITOR_H
#define IMAGEEDITOR_H
#include <qimage.h>
#include <qscrollview.h>
class ImageEditor : public QScrollView
{
    Q_OBJECT
    Q_PROPERTY(QColor penColor READ penColor WRITE setPenColor)
    Q_PROPERTY(QImage image READ image WRITE setImage)
    Q_PROPERTY(int zoomFactor READ zoomFactor WRITE setZoomFactor)
public:
    ImageEditor(QWidget *parent = 0, const char *name = 0);
    void setPenColor(const QColor &newColor);
    QColor penColor() const { return curColor;}
```

```
    void setZoomFactor(int newZoom);
    int zoomFactor() const { return zoom;}
    void setImage(const QImage &newImage);
    const QImage &image() const { return curImage;}
protected:
    void contentsMousePressEvent(QMouseEvent *event);
    void contentsMouseMoveEvent(QMouseEvent *event);
    void drawContents(QPainter *painter, int x, int y, int width, int height);
private:
    void drawImagePixel(QPainter *painter, int i, int j);
    void setImagePixel(const QPoint &pos, bool opaque);
    void resizeContents();
    QColor curColor;
    QImage curImage;
    int zoom;
};
#endif
```

머리부파일은 원래 클라스와 거의 비슷하다. 주요한 차이는 QWidget대신에 QScrollView로부터 계승하는것이다. 클라스의 실현을 고찰할 때 다른 차이도 고찰한다.

```
ImageEditor::ImageEditor(QWidget *parent, const char *name)
    : QScrollView(parent, name, WStaticContents | WNoAutoErase)
{
    curColor = black;
    zoom = 8;
    curImage.create(16, 16, 32);
    curImage.fill(qRgba(0, 0, 0, 0));
    curImage.setAlphaBuffer(true);
    resizeContents();
}
```

구성자는 WStaticContents와 WNoAutoErase기발들을 QScrollView에 넘긴다. 이 기발들은 실제로 보기구역에 설정된다. QScrollView의 기정값(Expanding,Expanding)이 적당하므로 크기암시를 설정하지 않는다.

원래 판에서는 Qt의 배치관리자들에 의하여 자체로 초기의 창문부품크기를 선택할수 있으므로 구성자에서 updateGeometry()를 호출하지 않았다. 그러나 여기서는 QScrollView기초클라스에 작업하려는 초기크기를 주어야 하고 resizeContents()호출에서 이것을 수행한다.

```
void ImageEditor::resizeContents()
{
    QSize size = zoom * curImage.size();
    if (zoom >= 3)
        size += QSize(1, 1);
    QScrollView::resizeContents(size.width(), size.height());
}
```

resizeContents()비공개함수는 QScrollView의 내용부분의 크기를 넘기여 QScrollView::resizeContents()를 호출한다. QScrollView는 보기구역이 내용부분의 어느 위치에 있는가에 따라 흘림띠들을 현시한다.

sizeHint()를 재정의할 필요는 없으며 QScrollView판은 내용의 크기를 리용하여 합리적인 크기암시를 제공한다.

```
void ImageEditor::setImage(const QImage &newImage)
{
    if (newImage != curImage) {
        curImage = newImage.convertDepth(32);
        curImage.detach();
        resizeContents();
        updateContents();
    }
}
```

IconEditor의 많은 원시함수들에서는 update()를 호출하여 재그리기를 진행하고 updateGeometry()를 호출하여 크기암시변경을 전달하였다. QScrollView판에서는 이 함수호출들이 내용의 크기변경에 대하여 QScrollView에 통지하는 resizeContents()와 재그리기하게 하는 updateContents()로 교체된다.

```
void ImageEditor::drawContents(QPainter *painter, int, int, int, int)
{
    if (zoom >= 3) {
        painter->setPen(colorGroup().foreground());
        for (int i = 0; i <= curImage.width(); ++i)
            painter->drawLine(zoom * i, 0, oom * i, zoom * curImage.height());
        for (int j = 0; j <= curImage.height(); ++j)
            painter->drawLine(0, zoom * j, zoom * curImage.width(), zoom * j);
    }
    for (int i = 0; i < curImage.width(); ++i) {
```

```
        for (int j = 0; j < curImage.height(); ++j)
            drawImagePixel(painter, i, j);
    }
}
```

drawContents()함수는 QScrollView에 의해 호출되며 내용의 구역을 다시그린다. QPainter객체는 이미 흘림변위를 계산하도록 초기화되여있다. 우리가 보통 paintEvent()에서와 같이 수행할 필요가 있다.

2~5번째 파라메터들은 다시 그려야 할 직4각형을 지정한다. 이 정보를 리용하여 다시 그려야 할 직4각형만 다시 그릴수 있지만 단순히 모두 다시 그린다.

drawContents()의 마감부근에서 호출되는 drawImagePixel()함수는 본질상 원래의 IconEditor클라스의 함수와 같으므로 여기서 다시 생성하지 않는다.

```
void ImageEditor::contentsMousePressEvent(QMouseEvent *event)
{
    if (event->button() == LeftButton)
        setImagePixel(event->pos(), true);
    else if (event->button() == RightButton)
        setImagePixel(event->pos(), false);
}
void ImageEditor::contentsMouseMoveEvent(QMouseEvent *event)
{
    if (event->state() & LeftButton)
        setImagePixel(event->pos(), true);
    else if (event->state() & RightButton)
        setImagePixel(event->pos(), false);
}
```

흘림보기의 내용부분들에 대한 마우스사건들은 QScrollView의 특수사건처리함수들을 재정의하여 조종할수 있으며 처리함수의 이름들은 모두 contents로 시작한다. 리면에서 QScrollView는 자동적으로 보기구역자리표들을 내용자리표들로 변환하므로 자체로 변환할 필요는 없다.

```
void ImageEditor::setImagePixel(const QPoint &pos, bool opaque)
{
    int i = pos.x() / zoom;
    int j = pos.y() / zoom;
    if (curImage.rect().contains(i, j)) {
        if (opaque)
```

```
                curImage.setPixel(i, j, penColor().rgb());
            else
                curImage.setPixel(i, j, qRgba(0, 0, 0, 0));
            QPainter painter(viewport());
            painter.translate(-contentsX(), -contentsY());
            drawImagePixel(&painter, i, j);
        }
    }
```

setImagePixel()함수는 contentsMousePressEvent()와 contentsMouseMoveEvent()함수로부터 호출되고 화소를 설정하거나 지운다. 코드는 원래의 판과 거의 같은데 QPainter객체를 초기화하는 방법이 다르다. 보기구역에서 그리기가 수행되므로 부모로서 viewport()를 넘기고 흘림변위에 맞게 QPainter의 자리표계를 변환한다.

QPainter를 취급하는 3개 행을 다음 행으로 교체한다.

```
updateContents(i * zoom, j * zoom, zoom, zoom);
```

이것은 QScrollView가 확대된 화상화소가 차지한 자그마한 직4각형만 갱신하도록 한다. 그러나 필요한 구역만 그리도록 drawContents()를 최적화하지 않았으므로 이것으로는 불충분하며 QPainter를 구성하고 그리기를 자체로 수행하는것이 더 좋다.

이제 ImageEditor를 사용한다면 원래판으로부터 QScrollView창문부품안에서 사용한 QWidget에 기초한 IconEditor를 원래판으로부터 실제로 구별할수 없다. 그러나 더 복잡한 창문부품들에서 QScrollView의 파생클라스화는 더 자연스러운 수법이다. 례를 들면 단어감기를 실현하는 QTextEdit와 같은 클라스는 표시하려는 문서와 QScrollView사이의 정확한 통합을 요구한다.

또한 내용이 너무 길거나 폭이 아주 넓으면 QScrollView의 파생클라스를 만들어야 한다. 그것은 일부 창문들이 32,767화소보다 더 큰 창문부품들을 유지하지 못하기때문이다.

ImageEditor실례에서 한가지 보여주지 못한것은 보기구역안에 자식창문부품들을 넣는것이다. 자식창문부품들은 단순히 addWidget()에 의해 추가되여야 하며 moveWidget()에 의해 옮길수 있다. 사용자가 내용구역을 흘림띠로 변화시킬 때마다 QScrollView는 자동적으로 화면우의 자식창문부품들을 옮긴다. (QScrollView가 많은 자식창문부품들을 포함한다면 흘림속도는 떠진다. 이 경우에 enableClipper(true)를 호출하여 최적화할수 있다.) 이 수법이 의의를 가지는 하나의 실례는 웨브열람기이다. 대부분의 내용은 보기구역에 직접 그려지지만 단추와 그밖의 폼입력요소들은 자식창문부품들로 표시된다.

제5절. 류동가능창문

류동가능창문(dock window)은 류동구역에서 류동할수 있는 창문이다. 도구띠는 류동창문의 기본실례이지만 다른 형태들도 있다.

QMainWindow는 4개의 류동구역을 제공한다. 즉 창문의 중심창문부품의 우, 아래, 왼쪽,

오른쪽에 각각 하나씩 있다. QToolBar들을 창조할 때 그것들은 자동적으로 그 부모의 류동구역안에 배치된다.

그림 6-11. 류동가능창문의 띄우기

매개 류동창문에는 손잡이가 있다. 이것은 그림 6-12에 보여주는 각 류동창문의 왼쪽과 우에 2개의 재색선들로 나타난다. 사용자들은 손잡이를 끌어서 한 류동구역에서 다른 류동구역으로 류동창문들을 이동할수 있다. 또한 류동창문을 구역에서 떼내여 류동구역밖으로 끌고감으로써 류동창문이 제일 웃준위창문으로 되게 할수 있다. 자유류동창문은 자기의 제목띠와 닫기단추를 가진다. 이런 창문들은 늘 자기 기본창문의 《제일 우에》 있다.

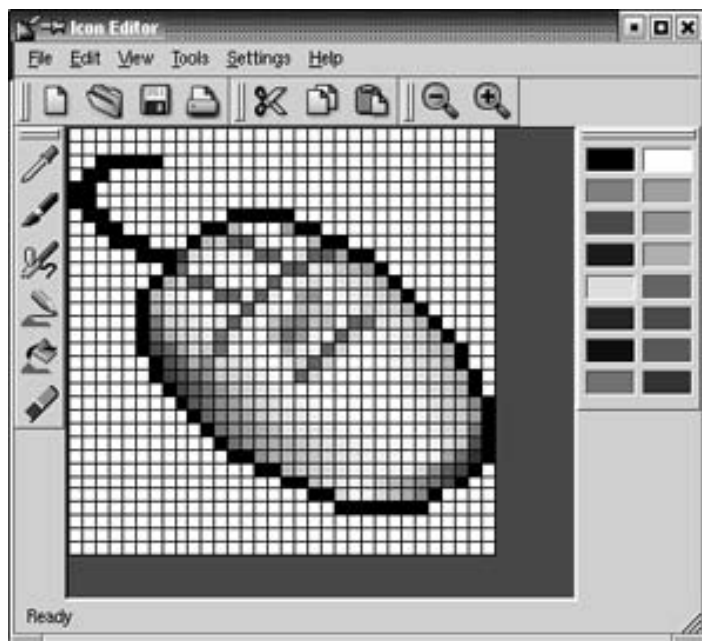



그림 6-12. 5개의 류동가능창문을 가지는 QMainWindow

류동창문이 떠오를 때 닫기단추를 삽입하려면 다음과 같이 setCloseMode()를 호출한다.

```
dockWindow->setCloseMode(QDockWindow::Undocked);
```

QDockArea는 모든 류동창문과 도구띠들의 목록을 가지고있는 문맥차림표를 제공한다. 류동창문이 일단 닫기면 사용자는 상황차림표를 리용하여 그것을 되살릴수 있다.

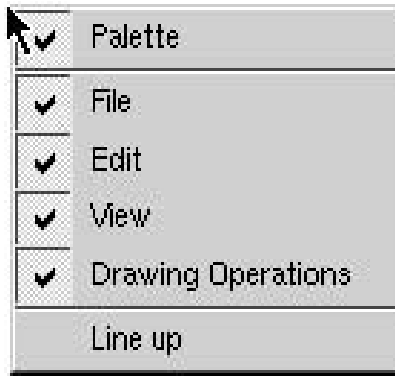



그림 6-13. QDockArea상황차림표

류동창문은 QDockWindow의 파생클라스이여야 한다. 단추들과 다른 창문부품들을 가지는 도구띠가 요구된다면 QDockWindow를 계승하는 QToolBar를 사용할수 있다. 여기에 QComboBox, QSpinBox, 여러개의 도구띠단추들을 포함하는 QToolBar를 창조하는 방법과 아래 류동구역에 그것을 배치하는 방법을 보여준 실례가 있다.

```
QToolBar *toolBar = new QToolBar(tr("Font"), this);
QComboBox *fontComboBox = new QComboBox(true, toolBar);
QSpinBox *fontSize = new QSpinBox(toolBar);
boldAct->addTo(toolBar);
italicAct->addTo(toolBar);
underlineAct->addTo(toolBar);
moveDockWindow(toolBar, DockBottom);
```

이 도구띠는 QComboBox와 QSpinBox가 너무 큰 수평공간을 요구하므로 사용자가 QMainWindow의 왼쪽이나 오른쪽 류동구역들로 도구띠를 이동한다면 좋지 않게 보일수 있다. 이러한 현상을 방지하기 위하여 QMainWindow::setDockEnabled()를 다음과 같이 호출할수 있다.

```
setDockEnabled(toolBar, DockLeft, false);
setDockEnabled(toolBar, DockRight, false);
```

필요한것이 류동창문부품이나 도구조색판과 같은것이라면 직접 QDockWindow를 리용하여 setWidget()를 호출하고 창문부품을 QDockWindow안에 표시되도록 설정할수 있다. 그러면 창문부품은 복잡해질수 있다. 사용자가 류동창문이 류동구역안에 있는 경우에도 류동창문의 크기를 조절하게 하려면 류동창문에서 setResizeEnabled()를 호출하면 된다. 그러면 류동창문의 측면에 분할기와 같은 손잡이가 나타나게 된다.

창문부품이 수평 혹은 수직류동구역안에 놓이는가에 따라서 자체를 변경하려고 한다면 QDockWindow::setOrientation()를 재정의하고 거기서 변경한다.

모든 도구띠와 다른 류동창문들의 위치를 보관하여 응용프로그람이 다시 실행될 때 그것들을 되살릴수 있게 하려면 QSplitter의 상태를 보관할 때 사용한 코드와 비슷하게

QMainWindow의 <<연산자로 상태를 써넣고 QMainWindow의 >>연산자로 그것을 다시 읽어들이는 코드를 쓸수 있다.

Microsoft Visual Studio와 Qt Designer와 같은 응용프로그람들은 류동창문들을 널리 리용하여 아주 유연한 사용자대면부를 제공한다. 보통 Qt에서 이러한 종류의 사용자대면부는 많은 사용자정의QDockWindow들과 함께 중간에 MDI자식창문들을 조종하기 위한 하나의 QWorkspace를 가지는 QMainWindow를 리용하여 작성한다.

제6절. 다중문서대면부

기본창문의 중심구역에 여러개의 문서를 제공하는 응용프로그람들은 다중문서대면부 응용프로그람이라고 부른다. Qt에서 MDI응용프로그람은 QWorkspace클라스에 의하여 중심창문부품으로서 창조되고 매개 문서창문을 QWorkspace의 자식으로 만든다.

보통 MDI응용프로그람은 창문들과 창문들의 목록을 관리하는 지령들을 포함하는 Windows차림표를 제공한다. 능동창문은 검사표식을 가지고 식별한다. 사용자는 Windows차림표에서 창문의 항목을 찰칵하여 그 창문을 능동으로 만들수 있다.

이 절에서는 그림 6-14에 보여주는 Editor응용프로그람을 개발하여 MDI응용프로그람을 창조하는 방법과 그 Windows차림표를 실현하는 방법을 보여준다.

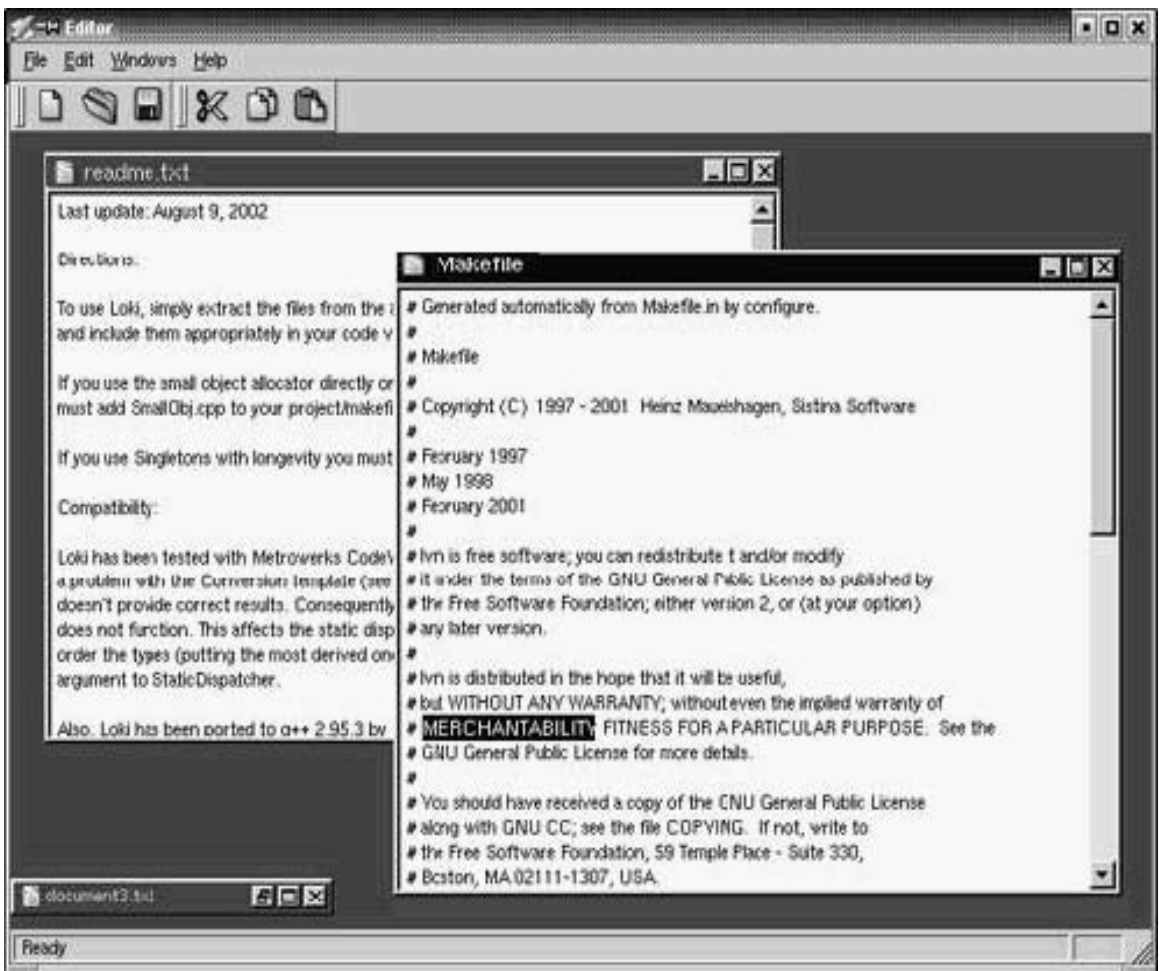



그림 6-14. Editor응용프로그람

응용프로그람은 2개의 클라스 즉 MainWindow와 Editor로 이루어진다. 코드의 대부분이 표계산프로그람과 같거나 비슷하므로 새 코드만 제시한다.

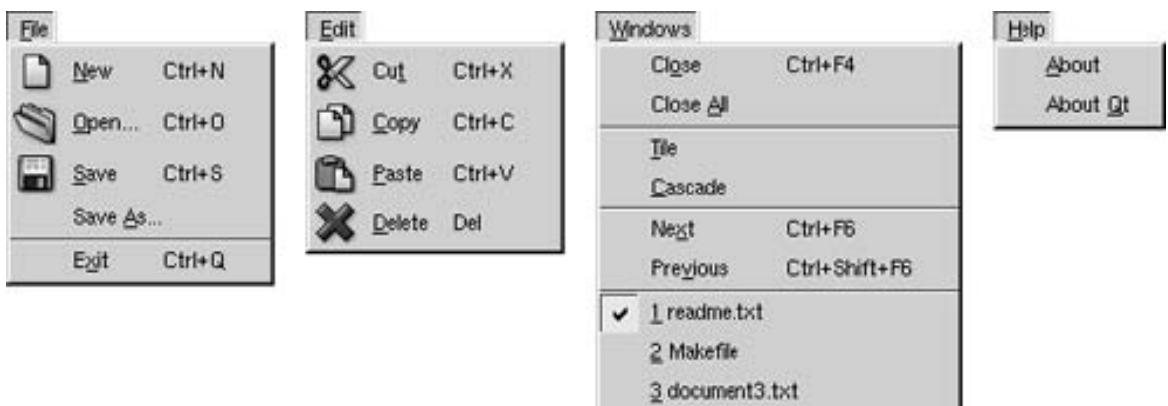



그림 6-15. Editor응용프로그람의 차림표들

MainWindow클라스로부터 시작하자.

```
MainWindow::MainWindow(QWidget *parent, const char *name) : QMainWindow(parent, name)
{
    workspace = new QWorkspace(this);
    setCentralWidget(workspace);
    connect(workspace, SIGNAL(windowActivated(QWidget *)), this, SLOT(updateMenus()));
    connect(workspace, SIGNAL(windowActivated(QWidget *)), this,
              SLOT(updateModIndicator()));
    createActions();
    createMenus();
    createToolBars();
    createStatusBar();
    setCaption(tr("Editor"));
    setIcon(QPixmap::fromMimeSource("icon.png"));
}
```

MainWindow구성자에서는 QWorkspace창문부품을 창조하여 중심창문부품으로 만든다. QWorkspace의 windowActivated()신호를 2개의 비공개처리부들에 련결한다. 이 처리부들은 차림표와 상태띠가 늘 현재 능동인 자식창문의 상태를 반영한다는것을 담보한다.

```
void MainWindow::newFile()
{
    Editor *editor = createEditor();
    editor->newFile();
    editor->show();
}
```

newFile()처리부는 File|New차림표선택에 대응된다. 이것은 createEditor()비공개함수에 따라서 자식Editor창문을 창조한다.

```
Editor *MainWindow::createEditor()
{
    Editor *editor = new Editor(workspace);
    connect(editor, SIGNAL(copyAvailable(bool)), this, SLOT(copyAvailable(bool)));
    connect(editor, SIGNAL(modificationChanged(bool)),this, SLOT(updateModIndicator()));
    return editor;
}
```

createEditor()함수는 Editor창문부품을 창조하고 2개의 신호–처리부련결을 설정한다. 첫째 련결은 Edit|Cut와 Edit|Copy가 선택된 본문이 있는가 없는가에 따라 허용 혹은 금지되도록 한다. 둘째 련결은 상태띠의 MOD지시자가 늘 최근의 상태를 반영하도록 한다.

MDI를 사용하므로 여러개의 Editor창문부품들을 사용할수 있다. 이것은 오직 능동인 Editor창문으로부터 오는 copyAvailable(bool)과 modificationChanged()신호들에 응답하는데만 관심을 가지는것과 관련된다. 그러나 이 신호들은 능동창문에만 발생될수 있으므로 실제로 문제는 없다.

```
void MainWindow::open()
{
    Editor *editor = createEditor();
    if (editor->open())
        editor->show();
    else
        editor->close();
}
```

open()함수는 File|Open에 대응된다. 이것은 새 문서에 대하여 새로운 Editor를 창조하고 Editor에 대하여 open()을 호출한다. 매개 Editor가 자체의 독립적인 상태를 관리해야 하므로 MainWindow클라스에서가 아니라 Editor클라스에서 파일조작을 실현하는것이 더 좋다. open()이 실패하면 사용자가 이미 오유를 통지받았으므로 편집기를 닫는다.

```
void MainWindow::save()
{
    if (activeEditor()) {
        activeEditor()->save();
        updateModIndicator();
    }
}
```

save()처리부는 능동편집기가 있으면 그것에 대하여 save()를 호출한다. 또한 실제의 작업을 수행하는 코드는 Editor클라스에 서술된다.

```
Editor *MainWindow::activeEditor()
{
    return (Editor *)workspace->activeWindow();
}
```

activeEditor()비공개함수는 능동자식창문을 Editor지적자로서 돌려준다.

```
void MainWindow::cut()
{
    if (activeEditor())
        activeEditor()->cut();
}
```

cut()처리부는 능동편집기에 대하여 cut()를 호출한다. copy(), paste(), del()처리부들도 같은 형태를 가진다.

```
void MainWindow::updateMenus()
{
    bool hasEditor = (activeEditor() != 0);
    saveAct->setEnabled(hasEditor);
    saveAsAct->setEnabled(hasEditor);
    pasteAct->setEnabled(hasEditor);
    deleteAct->setEnabled(hasEditor);
    copyAvailable(activeEditor() && activeEditor()->hasSelectedText());
    closeAct->setEnabled(hasEditor);
    closeAllAct->setEnabled(hasEditor);
    tileAct->setEnabled(hasEditor);
    cascadeAct->setEnabled(hasEditor);
    nextAct->setEnabled(hasEditor);
    previousAct->setEnabled(hasEditor);
    windowsMenu->clear();
    createWindowsMenu();
}
```

updateMenus()처리부는 창문이 능동으로 될 때마다(혹은 마지막 창문이 닫길 때) 호출되여 차림표체계를 갱신한다. 이때 MainWindow구성자에 배치한 신호–처리부련결을 리용한다.

대부분의 차림표선택은 능동창문이 있어야 의미를 가지므로 능동창문이 없으면 금지된다. 그다음 Windows차림표를 지우고 createWindowsMenu()를 호출하여 새로운 자식창문목록으로 다시 초기화한다.

```
void MainWindow::createWindowsMenu()
{
    closeAct->addTo(windowsMenu);
    closeAllAct->addTo(windowsMenu);
    windowsMenu->insertSeparator();
    tileAct->addTo(windowsMenu);
    cascadeAct->addTo(windowsMenu);
    windowsMenu->insertSeparator();
    nextAct->addTo(windowsMenu);
    previousAct->addTo(windowsMenu);
    if (activeEditor()) {
```

```
            windowsMenu->insertSeparator();
            windows = workspace->windowList();
            int numVisibleEditors = 0;
            for (int i = 0; i < (int)windows.count(); ++i) {
                QWidget *win = windows.at(i);
                if (!win->isHidden()) {
                    QString text = tr("%1 %2") .arg(numVisibleEditors + 1) .arg(win->caption());
                    if (numVisibleEditors < 9)
                        text.prepend("&");
                    int id = windowsMenu->insertItem( text, this, SLOT(activateWindow(int)));
                    bool isActive = (activeEditor() == win);
                    windowsMenu->setItemChecked(id, isActive);
                    windowsMenu->setItemParameter(id, i);
                    ++numVisibleEditors;
                }
            }
        }
    }
```

createWindowsMenu()비공개함수는 Windows차림표를 작용들과 보이는 창문들의 목록으로 채운다. 이 작용들은 모두 차림표의 전형으로서 QWorkspace의 closeActiveWindow(), closeAllWindows(), tile(), cascade()처리부들에 의해 간단히 실현된다.

능동창문의 항목은 그 이름옆에 검사표식이 있다. 사용자가 창문항목을 선택할 때 setItemParameter()호출에 의하여 activateWindow()처리부가 창문목록의 첨수를 파라메터로 하여 호출된다. 이것은 3장에서 표계산프로그람의 최근에 연 파일목록을 실현할 때와 아주 비슷하다.

처음 9개의 항목에 대하여 수자앞에 &기호를 배치하여 그 수의 한자리를 지름건으로 만든다. 다른 항목들에는 지름건을 주지 않는다.

```
    void MainWindow::activateWindow(int param)
    {
        QWidget *win = windows.at(param);
        win->show();
        win->setFocus();
    }
```

activateWindow()함수는 창문이 Windows차림표에서 닫길 때 호출된다. int파라메터는 setItemParameter()에 의해 설정되는 값이다. windows자료성원은 창문들의 목록을 보관하고

createWindowsMenu()에서 설정된다.

```
void MainWindow::copyAvailable(bool available)
{
    cutAct->setEnabled(available);
    copyAct->setEnabled(available);
}
```

copyAvailable()처리부는 편집기에서 본문을 선택하거나 선택이 해제될 때마다 호출된다. 또한 updateMenus()로부터 호출된다. 이것은 자르기와 복사작용을 허용하거나 금지한다.

```
void MainWindow::updateModIndicator()
{
    if (activeEditor() && activeEditor()->isModified())
        modLabel->setText(tr("MOD"));
    else
        modLabel->clear();
}
```

updateModIndicator()는 상태띠의 MOD지시자를 갱신한다. 이것은 편집기에서 본문이 수정될 때마다 호출된다. 또한 새 창문이 능동으로 될 때 호출된다.

```
void MainWindow::closeEvent(QCloseEvent *event)
{
    workspace->closeAllWindows();
    if (activeEditor())
        event->ignore();
    else
        event->accept();
}
```

closeEvent()함수는 모든 자식창문을 닫도록 재정의된다. 대체로 사용자가 "unsaved changes"통보창에서 취소하여 하나의 자식창문부품이 그의 닫기사건을 《무시》하면 MainWindow 의 닫기사건을 무시하고 그렇지 않으면 그것을 받아들이며 그 결과 Qt는 창문을 닫는다. MainWindow에서 closeEvent()를 재정의하지 않았으면 사용자에게는 보관하지 않은 변경을 보관할 기회가 주어지지 않는다.

이제는 MainWindow의 설명을 끝냈으므로 Editor실현에로 넘어갈수 있다. Editor클라스는 하나의 자식창문을 표시한다. 이것은 본문편집기능을 제공하는 QTextEdit를 계승한다. 임의의 Qt창문부품이 독립창문으로 쓰일수 있듯이 임의의 Qt창문부품을 MDI작업공간에서 자식창문으로 사용할수 있다.

여기에 클라스정의가 있다.

```
class Editor : public QTextEdit
{
    Q_OBJECT
public:
    Editor(QWidget *parent = 0, const char *name = 0);
    void newFile();
    bool open();
    bool openFile(const QString &fileName);
    bool save();
    bool saveAs();
    QSize sizeHint() const;
signals:
    void message(const QString &fileName, int delay);
protected:
    void closeEvent(QCloseEvent *event);
private:
    bool maybeSave();
    void saveFile(const QString &fileName);
    void setCurrentFile(const QString &fileName);
    QString strippedName(const QString &fullFileName);
    bool readFile(const QString &fileName);
    bool writeFile(const QString &fileName);
    QString curFile;
    bool isUntitled;
    QString fileFilters;
};
```

표계산프로그람의 MainWindow클라스에 있는 4개의 비공개함수 즉 maybeSave(), saveFile(), setCurrentFile(), strippedName()을 Editor클라스에서도 포함한다.

```
Editor::Editor(QWidget *parent, const char *name) : QTextEdit(parent, name)
{
    setWFlags(WDestructiveClose);
    setIcon(QPixmap::fromMimeSource("document.png"));
    isUntitled = true;
    fileFilters = tr("Text files (*.txt)\n" "All files (*)");
}
```

Editor구성자는 setWFlags()에 의하여 WDestructiveClose기발을 설정한다. 클라스구성자가 기발파라메터를 제공하지 않을 때(QTextEdit의 경우처럼) 여전히 setWFlags()에 의하여 대부분의 기발을 설정할수 있다.

사용자들이 임의의 개수의 편집기창문들을 창조할수 있으므로 편집기창문들의 이름지을 준비를 하여 처음으로 그것들을 보관하기전에 구별할수 있다. 이것을 취급하는 일반적인 방법의 하나는 수를 포함하는 이름을 할당하는것이다. (례를 들면 document1.txt). isUntitled변수를 리용하여 사용자가 제공한 이름들과 프로그람적으로 창조한 이름들을 구별한다.

구성자후에 newFile()이나 open()의 호출을 기대한다.

```
void Editor::newFile()
{
    static int documentNumber = 1;
    curFile = tr("document%1.txt").arg(documentNumber);
    setCaption(curFile);
    isUntitled = true;
    ++documentNumber;
}
```

newFile()함수는 새 문서에 대하여 document2.txt와 같은 이름을 생성한다. 코드는 구성자가 아니라 newFile()에 포함된다. 그것은 새로 창조한 Editor에서 현존 문서를 열기 위하여 open()을 호출할 때 문서번호를 증가시키려고 하지 않기때문이다. documentNumber가 static로 선언되므로 이것은 Editor의 모든 실례들에서 공유된다.

```
bool Editor::open()
{
    QString fileName = QFileDialog::getOpenFileName(".", fileFilters, this);
    if (fileName.isEmpty())
        return false;
    return openFile(fileName);
}
```

open()함수는 openFile()을 리용하여 현존파일을 연다.

```
bool Editor::save()
{
    if (isUntitled) {
        return saveAs();
    } else {
        saveFile(curFile);
        return true;
```

```
    }
}
```

save()함수는 isUntitled변수를 리용하여 saveFile()을 호출해야 하는가 saveAs()를 호출해야 하는가를 결정한다.

```
void Editor::closeEvent(QCloseEvent *event)
{
    if (maybeSave())
        event->accept();
    else
        event->ignore();
}
```

closeEvent()함수는 사용자가 보관안된 변경을 보관하도록 재정의된다. maybeSave()함수는 "Do you want to save your changes?"를 묻는 통보창을 펼친다. maybeSave()가 true를 돌려주면 닫기사건을 받아들이고 그렇지 않으면 그것을 무시하고 창문은 그대로 남아있다.

```
void Editor::setCurrentFile(const QString &fileName)
{
    curFile = fileName;
    setCaption(strippedName(curFile));
    isUntitled = false;
    setModified(false);
}
```

setCurrentFile()함수는 openFile()와 saveFile()로부터 호출되여 curFile와 isUntitled변수들을 갱신하고 창문제목을 설정하며 편집기의 《수정》기발을 false로 설정한다. Editor클라스는 QTextEdit로부터 setModified()와 isModified()를 계승하므로 자체의 변경기발을 관리할 필요는 없다. 사용자가 편집기의 본문을 수정할 때마다 QTextEdit는 modificationChanged()신호를 발생하고 그 내부수정기발을 true로 설정한다.

```
QSize Editor::sizeHint() const
{
    return QSize(72 * fontMetrics().width('x'), 25 * fontMetrics().lineSpacing());
}
```

sizeHint()함수는 문자 'x'의 폭과 본문행의 높이에 기초하는 크기를 돌려준다. QWorkspace는 크기암시를 사용하여 창문의 초기크기를 준다.

끝으로 Editor응용프로그람의 main.cpp파일을 보기로 하자.

```
#include <qapplication.h>
#include "mainwindow.h"
```

```
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    MainWindow mainWin;
    app.setMainWidget(&mainWin);
   if (argc > 1) {
      for (int i = 1;i < argc;++i)
         mainWin.openFile(argv[i]);
   } else {
      mainWin.newFile();
   }
   mainWin.show();
   return app.exec();
}
```

사용자가 지령행에서 임의의 파일들을 지정하면 그것들을 적재하려고 시도한다. 그렇지 않으면 빈 문서로 시작한다. -style 및 -font와 같은 Qt에 고유한 지령행선택은 QApplication구성자에 의해 인수목록으로부터 자동적으로 삭제된다. 그러므로 지령행에

```
editor -style=motif readme.txt
```

과 같이 쓰면 Editor응용프로그람은 하나의 문서 readme.txt를 가지고 기동한다.

MDI는 여러개의 문서를 동시에 취급하는 하나의 수법이다. 다른 수법은 여러개의 제일웃준위창문들을 리용하는것이다. 이 수법은 3장의 7절에서 설명한다.

제7장. 사건처리

GUI응용프로그람들은 사건구동형이다. 즉 응용프로그람이 기동하면 발생하는것은 사건의 결과이다. Qt로 프로그람을 작성할 때 어떤 일이 발생하면 Qt창문부품들이 신호들을 발생하므로 사건에 대하여 고찰할 필요가 있다. 사건은 자체의 사용자정의창문부품을 쓰거나 현존Qt창문부품들의 동작을 수정하려고 할 때 사용할수 있다.

이 장에서는 Qt의 사건모형을 설명한다. 우리는 Qt에서 각이한 형의 사건을 다루는 방법을 알게 된다. 또한 사건려과기를 리용하여 사건이 자기 목적지에 이르기전에 그것을 조종하는 방법을 고찰한다. 끝으로 Qt의 사건순환고리를 시험하고 긴장한 처리를 하는 동안에 사용자대면부가 응답하게 하는 방법을 설명한다.

제1절. 사건처리함수의 재정의

사건은 여러가지 정황에 따라서 창문체계나 Qt에 의해 생성된다. 사용자가 건사건이나 마우스단추를 누르거나 놓으면 건이나 마우스사건이 생성된다. 창문이 이동하여 가리워졌던 다른 창문이 로출될 때 그리기사건이 생성되여 새로 보이는 창문에 다시 그리기하여야 한다는것을 알려준다. 또한 사건은 창문부품이 건반초점을 얻거나 잃을 때마다 생성된다. 대부분의 사건은 사용자의 작용에 응답하여 생성되지만 시계사건 등 일부는 체계에 의해 독립적으로 생성된다.

사건을 신호와 혼돈하지 말아야 한다. 신호는 창문부품을 리용할 때 필요하지만 사건은 창문부품을 실현할 때 필요하다. 례를 들면 QPushButton을 사용하고있을 때 신호를 발생하는 저수준마우스사건이나 건사건보다도 그 clicked()신호에 더 관심을 가진다. 그러나 QPushButton과 같은 클라스를 실현하고있으면 마우스와 건사건들을 조종하고 필요할 때 clicked()신호를 발생하는 코드를 써야 한다.

사건은 QObject로부터 계승된 event()함수를 통하여 객체들에 통지된다. QWidget에서 event()실현은 mousePressEvent(), keyPressEvent(), paintEvent()와 같은 특정한 사건처리함수들에 가장 보편적인 형태이고 다른 종류의 사건들은 무시한다.

앞장들에서 MainWindow, IconEditor, Plotter, ImageEditor, Editor를 실현할 때 이미 많은 사건처리함수들을 보았다. 다른 형의 사건들이 많으며 QEvent참고문서에 렬거되여있다. 또한 사용자정의사건형들을 창조하고 사용자정의사건들을 자체로 발송할수 있다. 사용자정의사건들은 다중스레드응용프로그람들에서 특별히 편리하므로 17장(다중스레드작성)에서 설명한다. 여기서는 2가지 사건형 즉 건사건들과 시계사건들을 설명한다.

건사건들은 keyPressEvent()와 keyReleaseEvent()를 재정의하여 처리한다. Plotter창문부품은 keyPressEvent()를 재정의한다. 보통 놓기에서 중요한 건이란 수식건 Ctrl, Shift, Alt이고 이 건들은 state()를 사용하여 keyPressEvent()에서 검사할수 있으므로 keyPressEvent()를 재정의할 필요만 있다. 례를 들면 CodeEditor창문부품을 실현하는 경우에 Home과 Ctrl+Home사이를 구별하

는 keyPressEvent()는 다음과 같을수 있다.

```
void CodeEditor::keyPressEvent(QKeyEvent *event)
{
    switch (event->key()) {
        case Key_Home:
            if (event->state() & ControlButton)
                goToBeginningOfDocument();
            else goToBeginningOfLine();
                break;
        case Key_End: ...
        default:
            QWidget::keyPressEvent(event);
    }
}
```

Tab와 Backtab(Shift+Tab)건들은 특수한 경우이다. 이것들은 초점사슬에서 다음 또는 이전의 창문부품으로 초점을 넘긴다는 의미에서 keyPressEvent()를 호출하기전에 QWidget::event()에 의해 처리된다. 이 동작은 보통 경우에는 맞지만 CodeEditor창문부품에서는 Tab를 사용하여 행의 들여쓰기를 진행한다. 그때 event()재정의는 다음과 같다.

```
bool CodeEditor::event(QEvent * event)
{
    if (event->type() == QEvent::KeyPress) {
        QKeyEvent *keyEvent = (QKeyEvent *)event;
        if (keyEvent->key() == Key_Tab) {
            insertAtCurrentPosition('\t');
            return true;
        }
    }
    return QWidget::event(event);
}
```

사건이 건누르기이면 QEvent객체를 QKeyEvent로 강제변환하고 어느 건이 눌리웠는가를 검사한다. 건이 Tab이면 그에 대한 처리를 수행하고 true를 돌려주어 Qt에게 사건을 처리했다고 알린다. false를 돌려주었으면 Qt는 사건을 부모창문부품에 전달한다.

건속박을 실현하는 고급한 수법은 QAction을 사용하는것이다. 례를 들면 goToBeginningOfLine()과 goToBeginningOfDocument()가 CodeEditor창문부품에서 공개처리부들이고 CodeEditor가 MainWindow클라스의 중심창문부품으로 사용된다면 다음의 코드로 건결합을

추가할수 있다.

```
MainWindow::MainWindow(QWidget *parent, const char *name) : QMainWindow(parent, name)
{
    editor = new CodeEditor(this);
    setCentralWidget(editor);
    goToBeginningOfLineAct = new QAction(tr("Go to Beginning of Line"), tr("Home"), this);
    connect(goToBeginningOfLineAct, SIGNAL(activated()), editor,
            SLOT(goToBeginningOfLine()));
    goToBeginningOfDocumentAct = new QAction(tr("Go to Beginning of Document"),
            tr("Ctrl+Home"), this);
    connect(goToBeginningOfDocumentAct, SIGNAL(activated()), editor,
            SLOT(goToBeginningOfDocument()));
    ...
}
```

이것은 3장에서 본것처럼 차림표나 도구띠에 지령들을 간단히 추가하게 한다. 지령이 사용자대면부에 나타나지 않으면 QAction객체들은 건결합을 유지하기 위하여 내적으로 QAction에 의해 사용되는 클라스인 QAccel객체로 교체된다.

keyPressEvent()재정의와 QAction(혹은 QAccel)사용사이의 선택은 resizeEvent()재정의와 QLayout파생클라스사용사이의 선택과 비슷하다. QWidget의 파생클라스를 만들어 사용자정의창문부품을 실현한다면 일부 사건처리함수들을 재정의하고 거기에 동작코드를 간단히 쓸수 있다. 그러나 단지 창문부품을 사용한다면 QAction과 QLayout에 의해 제공되는 고수준대면부들이 더 편리하다.

또 하나의 일반사건은 시계사건이다. 대부분의 사건은 사용자작용의 결과로 발생하지만 시계사건은 응용프로그람이 규칙적인 시격으로 처리를 수행하게 한다. 시계사건은 유표의 깜빡거림이 없는 동화를 실현하거나 현시기를 초기화하는데 쓰일수 있다.

시계사건을 보여주기 위하여 Ticker창문부품을 실현한다. 이 창문부품은 30ms마다 1화소씩 왼쪽으로 흘러가는 본문띠(banner)를 표시한다. 창문부품의 폭이 본문보다 더 넓으면 본문은 창문부품의 전체폭을 채우도록 필요한만큼 반복된다.

sible to say ++ How long it lasted was impossible to say ++ Hov long it laste

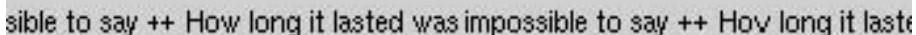



그림 7-1. Ticker창문부품

여기에 머리부파일이 있다.

```
#ifndef TICKER_H
#define TICKER_H
#include <qwidget.h>
```

```
class Ticker : public QWidget
{
    Q_OBJECT
    Q_PROPERTY(QString text READ text WRITE setText)
public:
    Ticker(QWidget *parent = 0, const char *name = 0);
    void setText(const QString &newText);
    QString text() const { return myText;}
    QSize sizeHint() const;
protected:
    void paintEvent(QPaintEvent *event);
    void timerEvent(QTimerEvent *event);
    void showEvent(QShowEvent *event);
    void hideEvent(QHideEvent *event);
private:
    QString myText;
    int offset;
    int myTimerId;
};
#endif
```

Ticker에서 4개의 사건처리함수를 재정의하고 그중 3개는 이전에 보지 못한것으로서 timerEvent(), showEvent(), hideEvent()이다.

그러면 실현을 고찰하자.

```
#include <qpainter.h>
#include "ticker.h"
Ticker::Ticker(QWidget *parent, const char *name) : QWidget(parent, name)
{
    offset = 0;
    myTimerId = 0;
}
```

구성자는 offset변수를 0으로 초기화한다. 본문을 그리는 x자리표는 offset값으로부터 끌어낸다.

```
void Ticker::setText(const QString &newText)
{
    myText = newText;
```

```
        update();
        updateGeometry();
    }
```

setText()함수는 표시하려는 본문을 설정한다. 이 함수는 update()를 호출하여 다시그리게 하고 updateGeometry()를 호출하여 Ticker창문부품에 응답할수 있는 배치관리자에 크기암시변경을 통지한다.

```
    QSize Ticker::sizeHint() const
    {
        return fontMetrics().size(0, text());
    }
```

sizeHint()함수는 본문이 요구하는 공간을 창문부품의 리상크기로서 돌려준다. QWidget::fontMetrics()함수는 창문부품의 서체관련정보를 얻기 위한 질문을 처리할수 있는 QFontMetrics객체를 돌려준다. 이 경우에 주어진 본문이 요구하는 크기를 묻는다.

```
    void Ticker::paintEvent(QPaintEvent *)
    {
        QPainter painter(this);
        int textWidth = fontMetrics().width(text());
        if (textWidth < 1)
            return;
        int x = -offset;
        while (x < width()) {
            painter.drawText(x, 0, textWidth, height(), AlignLeft | AlignVCenter, text());
            x += textWidth;
        }
    }
```

paintEvent()함수는 QPainter::drawText()에 의해 본문을 그린다. 이 함수는 fontMetrics()를 리용하여 본문이 요구하는 수평공간이 얼마나 되는가를 확정하고 필요한만큼 본문을 여러번 그리며 변위를 고려하여 창문부품의 전체폭을 완전히 채운다.

```
    void Ticker::showEvent(QShowEvent *)
    {
        myTimerId = startTimer(30);
    }
```

showEvent()함수는 시계를 기동한다. QObject::startTimer()호출은 ID번호를 돌려주는데 이것은 후에 시계를 식별하는데 사용된다. QObject는 자체의 시격을 가지는 여러개의 독립적인 시계들을 유지한다. startTimer()호출후에 Qt는 시계사건을 약 30ms간격으로 생성한다. 이때 정확

성은 기초하고있는 조작체계에 의존한다.

Ticker구성자에서 startTimer()를 호출할수 있으나 창문부품이 실제로 보일 때에만 Qt가 시계사건들을 생성하게 하여 자원을 절약할수 있다.

```
void Ticker::timerEvent(QTimerEvent *event)
{
    if (event->timerId() == myTimerId) {
        ++offset;
    if (offset >= fontMetrics().width(text()))
        offset = 0;scroll(-1, 0);
    } else {
        QWidget::timerEvent(event);
    }
}
```

timerEvent()함수는 체계에 의하여 같은 간격으로 호출된다. 시격은 본문폭까지의 이동을 모의하기 위해 변위를 1씩 증가시킨다. 그다음 QWidget::scroll()에 의하여 창문부품내용을 1화소 왼쪽으로 흘림한다. scroll()대신에 update()를 호출하면 충분한것 같지만 scroll()이 더 효과있고 깜빡거림을 방지한다. 그것은 scroll()이 화면우의 현존화소들을 단순히 이동하고 창문부품의 새로 로출된 구역(이 경우에 1화소폭의 띠)에 대하여서만 그리기사건을 생성하기때문이다.

만일 시계사건처리를 진행하지 않으려면 그것을 기초클라스에 넘긴다.

```
void Ticker::hideEvent(QHideEvent *)
{
    killTimer(myTimerId);
}
```

hideEvent()함수는 QObject::killTimer()를 호출하여 시계를 중지한다.

시계사건들은 저수준이고 여러개의 시계가 요구되면 시계ID들을 추적하여 보관하는것이 불편하다. 그러한 상황에 보통 매개 시계에 대하여 QTimer객체를 창조하는것이 더 편리하다. QTimer는 매 시격마다 timeout()신호를 발생한다. 또한 QTimer는 단일발사시계(한번만 시간을 요구하는 시계)에 편리한 대면부를 제공한다.

제2절. 사건려과기의 설치

Qt사건모형의 한가지 강력한 특성은 어떤 QObject실례가 자기 사건들을 알아보기전에 다른 QObject실례가 그 사건들을 감시하도록 설정할수 있는것이다.

여러개의 QLineEdit들로 구성된 CustomerInfoDialog창문부품이 있고 Space건으로 초점을 다음 QLineEdit로 옮기려고 한다고 하자. 이것은 QLineEdit의 파생클라스를 만들고 keyPressEvent()를 재정의하여 focusNextPrevChild()를 호출함으로써 실현할수 있다.

```
void MyLineEdit::keyPressEvent(QKeyEvent *event)
```

```
{
    if (event->key() == Key_Space)
        focusNextPrevChild(true);
    else
        QLineEdit::keyPressEvent(event);
}
```

이 수법에는 많은 결함이 있다. MyLineEdit가 표준Qt클라스가 아니므로 그것을 사용하는 폼을 설계하려면 Qt Designer에 통합되여야 한다. 또한 폼에서 각종 창문부품(례를 들면 QComboBoxe와 QSpinBoxe)들을 여러개 사용한다면 그것들을 파생클라스로 만들어 같은 동작을 보여주게 하고 Qt Designer에 통합하여야 한다.

더 좋은 해결책은 CustomerInfoDialog가 자식창문부품들의 건누르기사건들을 감시하게 하고 감시코드에 필요한 동작을 실현하는것이다. 이것은 사건려과기에 의해 달성된다. 사건려과기는 2단계에 걸쳐 설치한다.

① 목표에 대하여 installEventFilter()를 호출하여 목표객체와 함께 감시객체를 등록한다.

② 감시기의 eventFilter()함수에서 목표객체의 사건들을 처리한다.

감시객체를 등록하는 좋은 위치는 CustomerInfoDialog구성자이다.

```
CustomerInfoDialog::CustomerInfoDialog(QWidget *parent, const char *name)
        : QDialog(parent, name)
{
    ...
    firstNameEdit->installEventFilter(this);
    lastNameEdit->installEventFilter(this);
    cityEdit->installEventFilter(this);
    phoneNumberEdit->installEventFilter(this);
}
```

사건려과기가 등록되면 firstNameEdit, lastNameEdit, cityEdit, phoneNumberEdit창문부품들에 송신되는 사건들은 자기의 예정된 목적지에 송신되기전에 CustomerInfoDialog의 eventFilter()함수에 우선 송신된다. (여러개의 사건려과기가 같은 객체에 설치되여있으면 려과기들은 제일 최근에 설치된것부터 차례로 능동화된다.)

여기에 사건들을 받아들이는 eventFilter()함수가 있다.

```
bool CustomerInfoDialog::eventFilter(QObject *target, QEvent *event)
{
    if (target == firstNameEdit || target == lastNameEdit
                        || target == cityEdit || target == phoneNumberEdit) {
        if (event->type() == QEvent::KeyPress) {
```

```
            QKeyEvent *keyEvent = (QKeyEvent *) event;
            if (keyEvent->key() == Key_Space) {
               focusNextPrevChild(true);
               return true;
            }
         }
      }
      return QDialog::evenFilter(target, event);
   }
```

우선 목표창문부품이 QLineEdit들중 하나인가 검사한다. 기초클라스 QDialog는 자기의 창문부품들을 감시할수 있다. (Qt 3.2에서 이것은 QDialog의 경우가 아니다. 그러나 QMainWindow와 같은 다른 Qt창문부품클라스들은 여러가지 리유로 자기의 일부 자식창문부품들을 감시한다.)

사건이 건누르기이면 그것을 QKeyEvent로 강제변환하고 어느건을 눌렀는가 검사한다. 누른 건이 Space이면 focusNextPrevChild()를 호출하여 초점을 초점사슬의 다음 창문부품에 넘기고 true를 돌려주어 Qt에게 사건을 처리하였다는것을 알린다. false를 돌려주면 Qt는 사건을 예정한 목표에 보내고 결과 공백의 가상코드가 QLineEdit에 삽입된다.

사건이 Space건누르기가 아니면 조종을 eventFilter()의 기초클라스실현에 넘긴다.

Qt는 사건들을 처리하고 려과하는 5개의 준위를 제공한다.

① 특정한 사건처리함수를 재정의할수 있다.

mousePressEvent(), keyPressEvent(), paintEvent()와 같은 사건처리함수들의 재정의는 사건들을 처리하는 가장 일반적인 방법이다. 이미 그러한 실례를 여러개 보았다.

② QObject::event()를 재정의할수 있다.

event()함수를 재정의함으로써 사건이 특정한 사건처리함수들에 도달하기전에 처리된다. 이 수법은 처음에 보여준것처럼 Tab건의 기정의미를 무시하는데 대체로 필요하다. 이것은 또한 특정한 사건처리함수가 존재하지 않는 사건(례를 들면 LayoutDirectionChange)들처럼 드물게 나타나는 사건들을 처리하는데 쓰인다. event()를 재정의할 때 정확히 처리하지 못하는 경우를 취급하기 위하여 기초클라스의 event()함수를 호출해야 한다.

③ 하나의 QObject에 사건려과기를 설치할수 있다.

installEventFilter()에 의해 객체를 등록하였다면 목표객체의 모든 사건들은 우선 감시객체의 eventFilter()함수에 송신된다. 이 수법을 리용하여 우의 CustomerInfoDialog실례에서 Space건누르기를 처리하였다.

④ QApplication객체에 사건려과기를 설치할수 있다.

사건려과기가 qApp(유일한 QApplication객체)용으로 등록되였다면 응용프로그람안의 매개 객체에 대한 매개 사건은 다른 사건려과기에 송신되기전에 eventFilter()함수에 송신된다. 이

수법은 오유수정에 사용할수 있다. 또한 QApplication이 보통 무시하는 금지된 창문부품들에 송신된 마우스사건들을 처리하는데 쓰일수도 있다.

⑤ QApplication의 파생클라스를 만들고 notify()를 재정의할수 있다.

Qt는 QApplication::notify()를 호출하여 사건을 송신한다. 이 함수의 재정의는 사건려과기의 사건들을 볼 기회를 얻기전에 모든 사건들을 얻는 유일한 방법이다. 일반적으로 여러개의 사건려과기들이 동시에 있을수 있으나 notify()함수는 오직 하나이므로 사건려과기는 더 효과있다.

마우스와 건사건들을 비롯한 많은 사건형들을 전달할수 있다. 사건이 그 목표객체에로 가는 도중에 혹은 목표객체자체에 의하여 처리되지 않았다면 전체사건처리과정은 목표객체의 부모를 새 목표로 하여 반복된다. 이 과정은 사건이 처리되거나 제일 웃준위 객체에 이를 때까지 부모들을 따라 올라가면서 계속된다.

Caption
QDialog ❸
QGroupBox ❷
QCheckBox QCheckBox
QCheckBox QCheckBox ❶

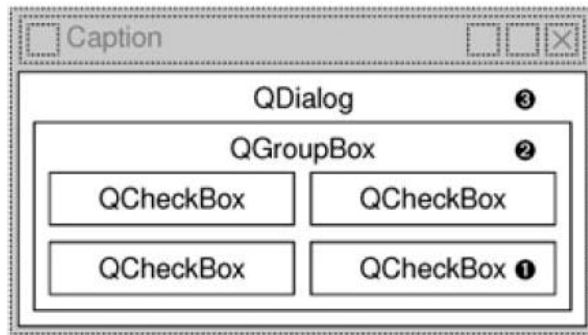



그림 7-2. 대화칸에서 사건전달

그림 7-2는 건누르기사건이 대화칸의 자식으로부터 부모에로 전달되여가는 방법을 보여준다. 사용자가 건을 누르면 사건은 우선 초점을 가지는 창문부품(이 경우에 오른쪽 아래의 QCheckBox)에로 송신된다. QCheckBox가 사건을 처리하지 않으면 Qt는 사건을 QGroupBox에 송신하고 끝으로 QDialog객체에 송신된다.

제3절. 응답성지연

QApplication::exec()를 호출할 때 Qt의 사건순환고리를 기동한다. Qt는 기동시에 몇가지 사건들을 발생하여 창문부품들을 표시하고 그린다. 그후에 사건순환고리를 실행하고 늘 사건이 발생하였는가 확인하고 사건들을 응용프로그람의 QObject들에 발송한다.

하나의 사건을 처리하는 도중에 다른 사건들이 생성되여 Qt의 사건대기렬에 추가될수 있다. 그러면 특정한 사건의 처리에 너무 많은 시간을 소비하고 사용자대면부의 반응이 떠진다. 례를 들면 응용프로그람이 파일을 디스크에 보관하는 동안에 창문체계에 의하여 생성된 사건들은 파일이 보관될 때까지 처리되지 않는다. 보관하는동안 응용프로그람은 창문체계로부터 자체를 다시 그리려는 요구에 응답하지 않는다.

하나의 해결책은 여러개의 스레드를 사용하는것이다. 즉 하나는 응용프로그람의 사용자대면부용 스레드이고 다른 하나는 파일보관(혹은 시간을 소비하는 다른 조작)을 수행하는 스레

드이다. 이렇게 응용프로그람의 사용자대면부는 파일을 보관하는동안 계속 응답성을 유지한다.(17장에서 이것을 달성하는 방법을 알게 된다.)

더 간단한 해결책은 파일보관코드에서 QApplication::processEvents()를 자주 호출하는것이다. 이 함수는 Qt가 대기하고있는 사건들을 처리하고 조종을 호출자에게 돌려준다. 사실상 QApplication::exec()는 while순환보다 processEvents()함수호출부근에 더 많다.

여기에 processEvents()를 리용하여 사용자대면부가 응답성을 지연하는 실례가 있다. 이 실례는 Spreadsheet의 파일보관코드에 기초하고있다.

```
bool Spreadsheet::writeFile(const QString &fileName)
{
    QFile file(fileName);
    ...
    for (int row = 0; row < NumRows; ++row) {
        for (int col = 0; col < NumCols; ++col) {
            QString str = formula(row, col);
            if (!str.isEmpty())
                out << (Q_UINT16)row << (Q_UINT16)col << str;
        }
        qApp->processEvents();
    }
    return true;
}
```

이 수법에서 한가지 위험한것은 현재 응용프로그람이 보관중에 있는데 사용자가 기본창문을 닫거나 지어는 *File|Save*를 찰칵하여 정의되지 않은 동작이 발생하는것이다. 이 문제를 해결하는 가장 간단한 방법은 다음의 호출

```
qApp->processEvents();
```

를 Qt에게 마우스와 건사건들을 무시하게 하는 다음의 호출로 바꾸는것이다.

```
qApp->eventLoop()->processEvents(QEventLoop::ExcludeUserInput);
```

흔히 오래동안 실행하는 조작이 발생하였을 때 QProgressDialog를 표시하려고 한다. QProgressDialog는 응용프로그람에 의해 이루어지는 진척상황에 대하여 사용자에게 통지하는 진척상황띠를 가지고있다. 또한 QProgressDialog는 사용자가 조작을 중지하게 하는 Cancel단추를 제공한다. 여기에 이 수법으로 Spreadsheet파일을 보관하는 코드가 있다.

```
bool Spreadsheet::writeFile(const QString &fileName)
{
    QFile file(fileName);
    ...
```

```
    QProgressDialog progress(tr("Saving file..."), tr("Cancel"), NumRows);
    progress.setModal(true);
    for (int row = 0;row < NumRows;++row) {
        progress.setProgress(row);
        qApp->processEvents();
        if (progress.wasCanceled()) {
            file.remove();
            return false;
        }
        for (int col = 0;col < NumCols;++col) {
            QString str = formula(row, col);
            if (!str.isEmpty())
                out << (Q_UINT16)row << (Q_UINT16)col << str;
        }
    }
    return true;
}
```

총결음수가 NumRows인 QProgressDialog를 창조한다. 그다음 각 행에 대하여 setProgress()를 호출하여 진척상황띠를 갱신한다. QProgressDialog는 자동적으로 현재 진척정형값을 총결음수로 나누어 퍼센트를 계산한다. QApplication::processEvents()를 호출하여 재그리기사건이나 사용자의 찰칵 혹은 건누르기를 처리한다. (례를 들면 사용자가 Cancel을 찰칵하게 한다.) 사용자가 Cancel을 찰칵하면 보관을 중지하고 파일을 삭제한다.

QProgressDialog에 대해서는 show()를 호출하지 않는다. 그것은 진척상황대화칸이 그 일을 자체로 수행하기때문이다. 보관하려는 파일이 작거나 콤퓨터가 고속이여서 조작이 짧은 시간에 끝난다면 QProgressDialog는 이것을 탐지하고 그자체를 전혀 표시하지 않는다.

실행시간이 긴 조작을 처리하는 완전히 다른 수법이 있다. 사용자가 요구할 때 처리를 수행하지 않고 응용프로그람이 무부하로 될 때까지 처리를 연기하는것이다. 이것은 응용프로그람이 얼마나 오래동안 무부하상태인가를 예견할수 없으므로 처리를 안전하게 중단하였다가 되살릴수 있다면 작업이 가능하다.

Qt에서 이 수법은 특수한 종류의 시계 즉 0ms시계를 리용하여 실현할수 있다. 이 시계들은 대기하고있는 사건이 없을 때 시간을 요구한다. 여기에 그 처리수법을 보여주는 timerEvent()실현의 실례가 있다.

```
void Spreadsheet::timerEvent(QTimerEvent *event)
{
    if (event->timerId() == myTimerId) {
```

```
            while (step < MaxStep && !qApp->hasPendingEvents()) {
                performStep(step);
                ++step;
            }
        } else {
            QTable::timerEvent(event);
        }
    }
```

hasPendingEvents()가 true를 돌려주면 처리를 중지하고 조종을 Qt에 돌려준다. 처리는 Qt가 대기하고있는 사건들을 모두 처리했을 때 되살아난다.

제8장. 2차원과 3차원도형처리

이 장에서는 Qt의 도형처리능력을 설명한다. Qt의 2차원그리기엔진은 QPainter로서 화면우의 창문부품, 화면밖의 픽스매프 혹은 인쇄기에 그리는데 사용될수 있다. 또한 Qt는 도형처리를 수행하는 고급한 방법을 제공하는 QCanvas클라스를 포함한다. 여기서는 여러가지 형태의 수천개 항목들을 효과적으로 처리할수 있는 항목에 기초한 수법을 리용한다. 미리 정의된 많은 항목들이 제공되며 사용자정의캔버스항목들을 쉽게 창조할수 있다.

QPainter와 QCanvas외에 또한 OpenGL서고를 사용하는 방법이 있다. OpenGL은 3차원도형을 그리기 위한 표준서고이지만 2차원도형그리기에도 사용할수 있다. Qt응용프로그람에 OpenGL코드를 통합하는것은 아주 간단한데 여기서 그것을 보여준다.

제1절. QPainter에 의한 그리기

QPainter는 창문부품이나 픽스매프와 같은 《그리기장치》에 그리기할 때 사용할수 있다. QPainter는 자체의 형식을 가지는 사용자정의창문부품들이나 사용자정의항목클라스들을 쓸 때 효과있다. 또한 QPainter는 인쇄에 사용되는 클라스로서 이 장의 뒤에서 자세히 설명한다.

QPainter는 기하학적도형들인 점, 선, 직4각형, 타원, 호, 현, 부채형, 다각형, 3차베쎌곡선을 그릴수 있다. 또한 픽스매프, 화상, 본문도 그릴수 있다.

QPainter구성자에 그리기장치를 넘길 때 QPainter는 장치로부터 일부 환경설정을 받아들이고 다른 환경설정을 기정값으로 설정한다. 이 설정은 그리기를 수행하는 방법에 영향을 준다. 3가지 가장 중요한것은 그리기장치의 펜, 솔, 서체이다.

· 펜은 직선과 기하학적도형의 테두리를 그리는데 사용된다. 펜은 색갈, 두께, 선형식, 모자(cap)형식, 결합형식으로 이루어진다.

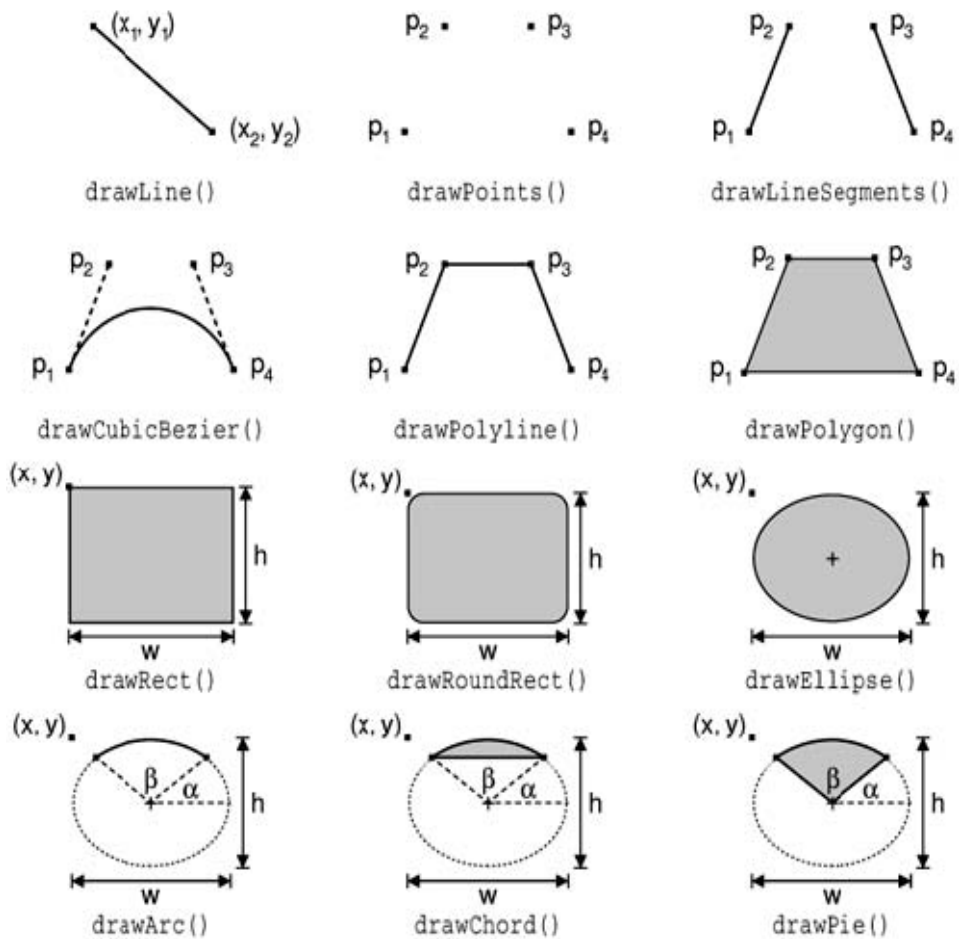



그림 8-1. 기하학적도형을 그리는 QPainter함수들

그림 8-2. 펜의 형식들

· 솔은 기하학적도형을 채우는데 사용되는 패턴이다. 솔은 색갈과 형식으로 이루어진다.

· 서체는 본문을 그리는데 쓰인다. 서체는 계렬과 점크기 등 많은 속성들을 가지고있다.

이러한 설정은 QPen, QBrush 혹은 QFont객체에서 setPen(), setBrush(), setFont()를 호출하여 변경할수 있다.

FlatCap SquareCap RoundCap

MiterJoin BevelJoin RoundJoin

그림 8-3. 모자형식과 결합형식들

SolidPattern Dense1Pattern Dense2Pattern Dense3Pattern Dense4Pattern

Dense5Pattern Dense6Pattern Dense7Pattern HorPattern VerPattern

CrossPattern BDiagPattern FDiagPattern DiagCross-Pattern NoBrush

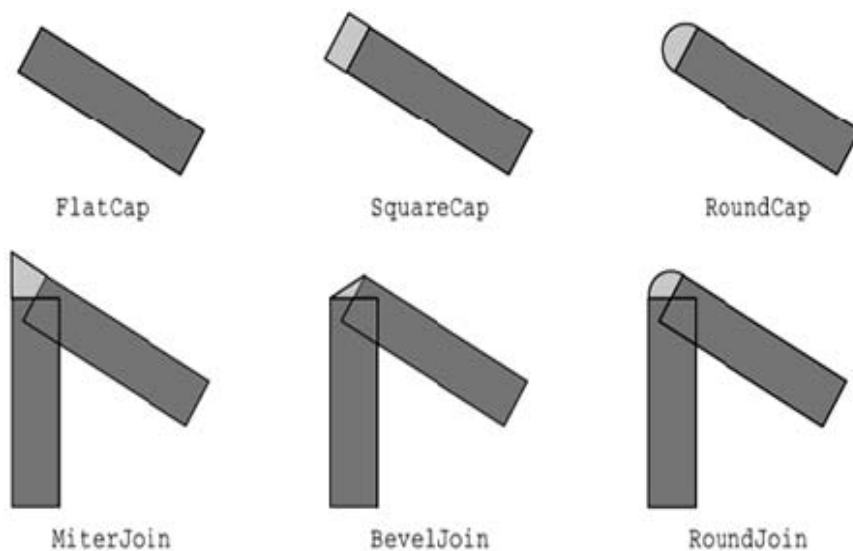
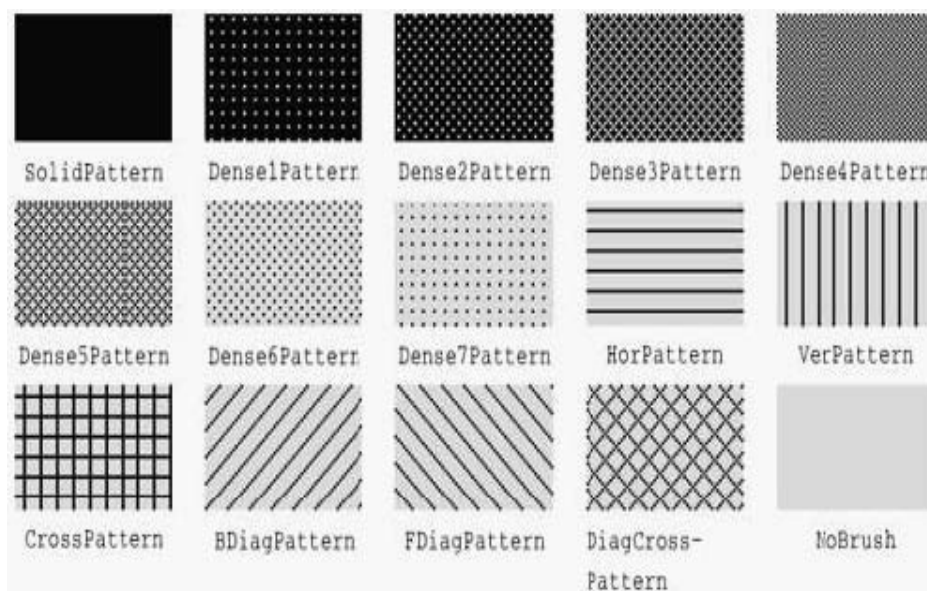



그림 8-4. 솔의 형식들

여기에 그림 8-5(ㄱ)에서 보여준 타원을 그리는 코드가 있다.

ㄱ) 타원 ㄴ) 부채형 ㄷ) 베쩰곡선

그림 8-5. 기하학적도형의 실례들

```
QPainter  painter(this);
painter.setPen(QPen(black,  3,  DashDotLine));
painter.setBrush(QBrush(red,  SolidPattern));
painter.drawEllipse(20, 20, 100, 60);
```

여기에 그림 8-5(ㄴ)에서 보여준 부채형을 그리는 코드가 있다.

```
QPainter  painter(this);
painter.setPen(QPen(black,  5,  SolidLine));
painter.setBrush(QBrush(red,  DiagCrossPattern));
painter.drawPie(20, 100, 60, 60 * 16, 270 * 16);
```

drawPie()의 마지막 2개 인수는 시작각도와 마감각도를 16배한것이다.

여기에 그림 8-5(ㄷ)에서 보여준 3차원베쎌곡선을 그리는 코드가 있다.

```
QPainter  painter(this);
QPointArray  points(4);
points[0]  =  QPoint(20,  80);
points[1]  =  QPoint(50,  20);
points[2]  =  QPoint(80,  20);
points[3]  =  QPoint(120,  80);
painter.setPen(QPen(black,  3,  SolidLine));
painter.drawCubicBezier(points);
```

그리기장치의 현재 상태는 save()호출에 의해 탄창에 보관되고 후에 restore()를 호출할 때 되살아난다. 이것은 그리기장치의 설정을 일시 변경하였다가 이전값들로 재설정하려고 할 때 쓸모있다.

펜, 솔, 서체외에 그리기장치를 조종하는 다른 설정은 다음과 같다.

· 배경색은 배경방식이 OpaqueMode(기정값은 TransparentMode)일 때 기하도형, 본문 혹은 비트매프의 배경을 솔패턴으로 채우는데 리용한다.

· 라스터조작(raster operation)은 그리기장치에 이미 표시되여있는 화소들과 새로 그려지는 화소들을 결합하는 방법을 지정한다. 기정값은 CopyROP로서 새 화소가 이전 화소값을 무시하고 장치에 단순히 복사된다는것을 의미한다. 다른 라스터조작으로서 XorROP, NotROP, AndROP, NotAndROP가 있다.

· 솔원점은 솔패턴의 시작점으로서 보통 창문부품의 왼쪽웃구석이다.

· 잘라내기령역(clip region)은 그리기할수 있는 장치의 구역이다. 잘라내기령역밖에서 수행한 그리기조작은 무시된다.

· 보기구역, 창문, 세계행렬(world matrix)은 론리QPainter자리표를 물리그리기장치자리표로 넘기는 방법을 결정한다. 기정으로 론리와 물리자리표계들은 일치하도록 설정된다.

보기구역, 창문, 세계행렬에 의하여 정의된 자리표계를 더 구체적으로 고찰하자. (이 상황에서 《창문》이라는 용어는 제일 웃준위창문부품의 의미에서 창문을 말하는것이 아니며 또 《보기구역》은 QScrollView의 보기구역에서 수행해야 하는 일을 가지고있는것이 아니다.)

보기구역과 창문은 정확히 한계가 있다. 보기구역(viewport)은 물리자리표로 지정된 임의의 직4각형이다. 창문(window)은 같은 직4각형을 지정하지만 론리자리표로 되여있다. 그리기할 때에는 점들을 론리자리표로 지정하고 이 자리표들은 현재의 창문–보기구역설정에 기초하는 선형대수적방법에 의하여 물리자리표로 변환된다.

기정으로 보기구역과 창문은 장치의 직4각형으로 설정된다. 례를 들면 장치가 320×200창문부품이면 보기구역과 창문은 꼭같이 왼쪽웃구석의 위치가 (0, 0)인 320×200직4각형이다. 이 경우에 론리자리표계와 물리자리표계는 같다.

창문–보기구역기구는 그리기장치의 크기나 분해능에 의존하지 않는 그리기코드를 작성하는데 쓸모있다. 우리는 항상 산수적으로 론리자리표를 물리자리표로 넘길수 있지만 QPainter가 그렇게 하도록 하는것이 좋다. 례를 들면 중심이 (0, 0)인 론리자리표를 (-50, -50)으로부터 (+50, +50)까지 전개하려고 한다면 창문을 다음과 같이 설정할수 있다.

```
painter.setWindow(QRect(-50, -50, 100, 100));
```

(-50, -50)쌍은 원점을 지정하고 (100, 100)쌍은 폭과 높이를 지정한다. 이것은 론리자리표 (-50, -50)이 현재 물리자리표(0, 0)에 대응되고 론리자리표 (+50, +50)이 물리자리표 (320, 200)에 대응된다는것을 의미한다. 이 실례에서 보기구역을 변경할 필요는 없다.

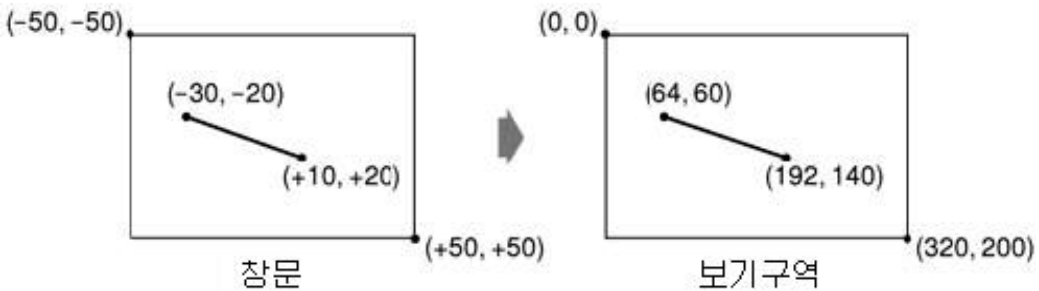



그림 8-6. 론리자리표를 물리자리표로 변환

그러면 세계행렬을 고찰하자. 세계행렬은 창문–보기구역변환과 함께 적용되는 변환행렬이다. 이것은 그리고있는 항목들을 변환하고 신축하고 회전하거나 자르게 한다. 례를 들면 본문을 45°각으로 그리려고 한다면 다음의 코드를 사용할수 있다.

```
QWMatrix  matrix;
matrix.rotate(45.0);
painter.setWorldMatrix(matrix);
painter.drawText(rect,  AlignCenter,  tr("Revenue"));
```

drawText()에 넘기는 론리자리표들은 세계행렬로 변환된 다음 창문–보기구역설정을 리용하여 물리자리표로 넘어간다.

다중변환을 지정하면 변환들이 주어지는 차례로 적용된다. 례를 들면 회전축점으로서 점

(10, 20)을 리용하려고 한다면 창문을 변환하고 회전하고 창문을 원래 위치로 다시 변환한다.

```
QWMatrix matrix;
matrix.translate(-10.0, -20.0);
matrix.rotate(45.0);
matrix.translate(+10.0, +20.0);
painter.setWorldMatrix(matrix);
painter.drawText(rect, AlignCenter, tr("Revenue"));
```

변환을 지정하는 더 간단한 방법은 QPainter의 translate(), scale(), rotate(), shear()편의함수들을 사용하는것이다.

```
painter.translate(-10.0, -20.0);
painter.rotate(45.0);
painter.translate(+10.0, +20.0);
painter.drawText(rect, AlignCenter, tr("Revenue"));
```

그러나 같은 변환을 반복하려고 한다면 변환들을 QWMatrix객체에 보관하고 변환이 필요할 때마다 그리기장치에 세계행렬을 설정하는것이 더 빠르다.

세계행렬을 보관하였다가 후에 되살리려면 saveWorldMatrix()와 restoreWorldMatrix()를 리용할수 있다.

그리기장치변환을 설명하기 위하여 그림 8-7에 보여주는 OvenTimer창문부품의 코드를 고찰한다. OvenTimer창문부품은 내부에 시계를 가지고있는 로에서 일반적으로 사용하고있는 물리적인 로(oven)시계를 모형화한것이다. 사용자는 눈금(notch)을 찰칵하여 지속시간을 설정할수 있다. 바퀴는 자동적으로 0에 이를 때까지 시계바늘과 반대방향으로 돌아가며 0점에서 OvenTimer는 timeout()신호를 발생한다.

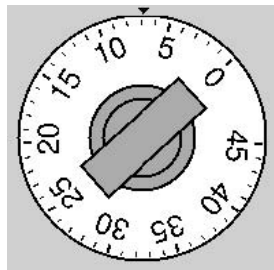



그림 8-7. OvenTimer창문부품

```
class OvenTimer : public QWidget
{
    Q_OBJECT
public:
    OvenTimer(QWidget *parent, const char *name = 0);
    void setDuration(int secs);
```

```
    int duration() const;
    void draw(QPainter *painter);
signals:
    void timeout();
protected:
    void paintEvent(QPaintEvent *event);
    void mousePressEvent(QMouseEvent *event);
private:
    QDateTime finishTime;
    QTimer *updateTimer;
    QTimer *finishTimer;
};
```

OvenTimer클라스는 QWidget를 계승하며 2개의 가상함수 paintEvent()와 mousePressEvent()를 재정의한다.

```
#include <qpainter.h>
#include <qpixmap.h>
#include <qtimer.h>
#include <cmath>
using namespace std;
#include "oventimer.h"
const double DegreesPerMinute = 7.0;
const double DegreesPerSecond = DegreesPerMinute / 60;
const int MaxMinutes = 45;
const int MaxSeconds = MaxMinutes * 60;
const int UpdateInterval = 10;
OvenTimer::OvenTimer(QWidget *parent, const char *name) : QWidget(parent, name)
{
    finishTime = QDateTime::currentDateTime();
    updateTimer = new QTimer(this);
    finishTimer = new QTimer(this);
    connect(updateTimer, SIGNAL(timeout()), this, SLOT(update()));
    connect(finishTimer, SIGNAL(timeout()), this, SIGNAL(timeout()));
}
```

구성자에서는 2개의 QTimer객체를 창조하는데 updateTimer는 창문부품의 모양을 규칙적인 시격으로 갱신하는데 쓰이고 finishTimer는 시계가 0에 이를 때 창문부품의 timeout()신호를 발

생한다.

```
void OvenTimer::setDuration(int secs)
{
    if (secs > MaxSeconds)
        secs = MaxSeconds;finishTime = QDateTime::currentDateTime().addSecs(secs);
    updateTimer->start(UpdateInterval * 1000, false);
    finishTimer->start(secs * 1000, true);
    update();
}
```

setDuration()함수는 로시계의 지속시간을 주어진 초수로 설정한다. updateTimer의start()호출에 넘기는 false인수는 Qt에 이것이 10s간격으로 시간을 요구하는 반복시계라는것을 말해준다. finishTimer는 한번 시간을 요구하는데 필요하므로 true인수를 리용하여 그것이 단일발사시계라는것을 가리킨다. 초단위의 지속시간을 QDateTime::currentDateTime()에 의해 얻어진 현재 시간에 더하여 완료시간을 계산하고 finishTime비공개변수에 보관한다.

finishTime변수는 날자와 시간을 보관하는 Qt자료형인 QDateTime형이다. QDateTime의 date요소는 현재시간이 자정전이고 완료시간이 자정후인 경우에 중요하다.

```
int OvenTimer::duration() const
{
    int secs = QDateTime::currentDateTime().secsTo(finishTime);
    if (secs < 0)
        secs = 0;
    return secs;
}
```

duration()함수는 시계가 완료하기전에 남은 초수를 돌려준다.

```
void OvenTimer::mousePressEvent(QMouseEvent *event)
{
    QPoint point = event->pos() -rect().center();
    double theta = atan2(-(double)point.x(), -(double)point.y()) * 180 / 3.14159265359;
    setDuration((int)(duration() + theta / DegreesPerSecond));
    update();
}
```

사용자가 창문부품을 찰칵하면 효과적인 수학식을 리용하여 가장 가까운 눈금을 찾아내고 결과를 리용하여 새 지속시간을 설정한다. 그다음 재그리기를 발생한다. 사용자가 찰칵한 눈금은 현재 제일 우에 있으며 0에 이를 때까지 시계바늘방향으로 이동한다.

```
void OvenTimer::paintEvent(QPaintEvent *)
```

```
{
    QPainter painter(this);
    int side = QMIN(width(), height());
    painter.setViewport((width() -side) /2, (height() -side) /2, side, side);
    painter.setWindow(-50, -50, 100, 100);
    draw(&painter);
}
```

paintEvent()에서는 보기구역을 창문부품에 알맞는 최대바른4각형구역으로 설정하고 창문을 직4각형(-50, -50, 100, 100) 즉 (-50, -50)으로부터 (+50, +50)범위의 100×100 직4각형으로 설정한다. QMIN()마크로는 2개 인수의 최소값을 돌려준다.

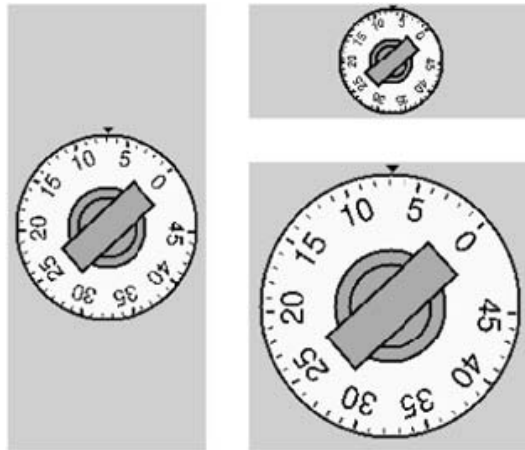



그림 8-8. 3가지 다른 크기의 OvenTimer창문부품

보기구역을 바른4각형으로 설정하지 않았으면 로시계는 창문부품이 바른4각형이 아닌 직4각형으로 크기가 달라질 때 타원으로 된다. 일반적으로 그러한 변형을 피하려면 보기구역과 창문을 같은 가로세로비를 가지는 직4각형들로 설정해야 한다.

또한 (-50, -50, 100, 100)의 창문은 아래와 같은 원인을 고려하여 설정되였다.

· QPainter의 draw함수들은 int자리표값들을 가진다. 창문을 너무 작게 선택하면 필요한 모든 점을 옹근수로 지정할수 없다.

· 큰 창문을 사용하고 drawText()을 리용하여 본문을 그리려면 그것을 보상하는데 더 큰 서체가 요구된다.

이것은 말하자면 (-5, -5, 10, 10) 혹은 (-2000, -2000, 4000, 4000)보다 (-50, -50, 100, 100)이 더 좋은 선택으로 되게 한다.

그러면 그리기코드를 고찰하자.

```
void OvenTimer::draw(QPainter *painter)
{
    static const QCOORD triangle[3][2] ={
```

```
    { -2, -49 }, { +2, -49 }, { 0, -47 }
};
QPen thickPen(colorGroup().foreground(), 2);
QPen thinPen(colorGroup().foreground(), 1);
painter->setPen(thinPen);
painter->setBrush(colorGroup().foreground());
painter->drawConvexPolygon(QPointArray(3, &triangle[0] [0]));
```

창문부품의 꼭대기에 0위치를 표식하는 아주 작은 3각형을 그리는것으로 시작한다. 3각형을 3개의 고정자리표들로 지정하고 drawConvexPolygon()을 리용하여 그린다. drawPolygon()을 리용할수 있으나 그리고있는 다각형이 불록하다는것을 알고있을 때 drawConvexPolygon()을 호출하여 조금이나마 시간을 절약할수 있다.

창문–보기구역기구가 아주 편리한것은 그리기지령들에서 사용하는 자리표들을 고정코드화하여도 좋은 크기조절동작을 얻을수 있는것이다. 바른4각형이 아닌 창문부품들에 대하여 걱정하지 않아도 보기구역을 적당히 설정하여 처리한다.

```
painter->setPen(thickPen);
painter->setBrush(colorGroup().light());
painter->drawEllipse(-46, -46, 92, 92);
painter->setBrush(colorGroup().mid());
painter->drawEllipse(-20, -20, 40, 40);
painter->drawEllipse(-15, -15, 30, 30);
```

바깥원과 2개의 아낙원을 그린다. 바깥원은 조색판의 《밝은》 요소(일반적으로 백색)로 채우고 아낙원들은 《중간》요소(일반적으로 중간재색)로 채운다.

```
int secs = duration();
painter->rotate(secs * DegreesPerSecond);
painter->drawRect(-8, -25, 16, 50);
for (int i = 0;i <= MaxMinutes;++i) {
    if (i % 5 == 0) {
        painter->setPen(thickPen);
        painter->drawLine(0, -41, 0, -44);
        painter->drawText(-15, -41, 30, 25, AlignHCenter | AlignTop, QString::number(i));
    } else {
        painter->setPen(thinPen);
        painter->drawLine(0, -42, 0, -44);
    }
    painter->rotate(-DegreesPerMinute);
```

```
        }
    }
```

손잡이, 눈금 그리고 5번째의 매개 눈금마다 분을 그린다. rotate()를 호출하여 그리기장치의 자리표계를 회전한다. 낡은 자리표계에서 0분표식은 제일우에 있고 현재 0분표식은 나머지 시간에 적합한 위치로 이동된다. 직4각형손잡이의 방향이 회전각도에 의존하므로 회전후에 손잡이를 그린다.

for순환으로 바깥원의 둘레를 따라 눈금표식을 새기고 5분마다 수자들을 새겨넣는다. 본문은 눈금표식아래의 보이지 않는 직4각형안에 넣는다. 한번 순환의 끝에서 그리기장치를 1분에 대응하는 량인 7°씩 시계바늘방향으로 회전한다. drawLine()과 drawText()호출에 넘기는 자리표들이 늘 갈다할지라도 다음에 눈금표식을 그릴 때 원둘레의 각이한 위치에 있게 된다.

로시계를 실현하는 또 하나의 방법은 sin()과 cos()를 리용하여 (*x,y*)위치를 자체로 계산하여 원둘레에서의 위치들을 찾는것이다. 그러나 그때 여전히 변환과 회전을 리용하여 어떤 각도로 본문을 그릴수 있다.

한가지 문제는 10s마다 창문부품전체를 다시 그리기하는데 그때마다 깜빡거림이 발생하는것이다. 해결책은 2중완충기능을 추가하는것이다. 2중완충은 WNoAutoErase를 기초클라스구성자에 넘기고 처음에 보여준 paintEvent()함수를 다음것과 교체하여 수행할수 있다.

```
void OvenTimer::paintEvent(QPaintEvent *event)
{
    static QPixmap pixmap;
    QRect rect = event->rect();
    QSize newSize = rect.size().expandedTo(pixmap.size());
    pixmap.resize(newSize);
    pixmap.fill(this, rect.topLeft());
    QPainter painter(&pixmap, this);
    int side = QMIN(width(), height());
    painter.setViewport((width() -side) / 2 -event->rect().x(),
                (height() -side) / 2 -event->rect().y(), side, side);
    painter.setWindow(-50, -50, 100, 100);
    draw(&painter);
    bitBlt(this, event->rect().topLeft(), &pixmap);
}
```

이번에는 창문부품대신에 픽스매프에 직접 그린다. 픽스매프는 다시 그리려는 구역의 크기로 주어지고 창문–보기구역쌍은 창문부품에 직접 수행되는것처럼 그리기가 수행되도록 초기화된다. draw()함수도 역시 변경되지 않는다. 끝으로 bitBlt()를 리용하여 창문부품에 픽스매프를 복사한다.

이것은 5장의 4절에서 설명한것과 비슷하지만 한가지 중요한 차이가 있다. 5장에서 translate()를 리용하여 그리기장치를 변환하였으나 여기서는 보기구역을 설정할 때 그리기사건의 *x*와 *y*자리표들을 던다. 여기서 변환의 리용은 변환이 론리창문자리표로 표시되여야 하므로 편리하지 않지만 사건의 직4각형은 물리자리표로 되여있다.

제2절. QCanvas에 의한 도형처리

QCanvas는 QPainter가 제공하는것보다 도형처리를 수행하기 위한 더 고급한 대면부를 제공한다. QCanvas는 임의의 형태의 항목들을 포함하며 내적으로 2중완충을 리용하여 깜빡거림을 피한다. 자료시각화프로그람들과 2차원유희와 같이 사용자가 조작할수 있는 항목들을 표시할 필요가 있는 응용프로그람들에서 QCanvas의 사용은 QWidget::paintEvent()나 QScrollView::drawContents()를 재정의하고 모든것을 수동적으로 다시 그리는것보다 더 좋은 수법이다.

QCanvas에 보여준 항목들은 QCanvasItem이나 그 파생클라스의 실례들이다. Qt는 미리 정의된 파생클라스들의 모임을 제공한다. 즉 QCanvasLine, QCanvasRectangle, QCanvasPolygon, QCanvasPolygonalItem, QCanvasEllipse, QCanvasSpline, QCanvasSprite, QCanvasText들을 제공한다. 이 클라스들은 그 자체가 사용자정의캔버스항목들을 제공하는 파생클라스로 만들수 있다.

QCanvas와 그의 QCanvasItem들은 순수 자료이며 시각적표시를 가지지 않는다. 캔버스와 그 항목들을 표시하려면 QCanvasView창문부품을 사용해야 한다. 이와 같이 자료와 그 시각적 표시의 분리는 같은 캔버스를 시각화하는 여러개의 QCanvasView창문부품들을 가질수 있게 한다. 매개의 QCanvasView는 될수록 각이한 변환행렬을 가지고 캔버스의 자기부분을 표시할수 있다.

QCanvas는 대량의 항목들을 처리하도록 고도로 최적화되여있다. 항목이 달라질 때 QCanvas는 달라진 부분을 다시 그린다. 또한 효과적인 충돌탐색알고리듬을 제공한다. 이러한 리유로 QWidget::paintEvent() 혹은 QScrollView::drawContents()를 재정의하여 QCanvas를 고찰한다.

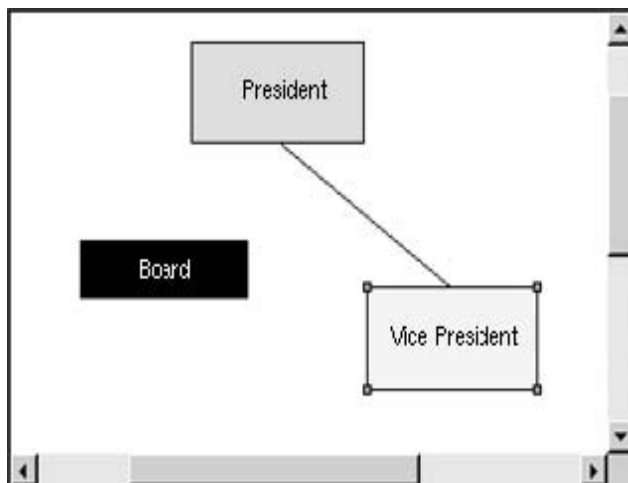



그림 8-9. DiagramView창문부품

QCanvas사용법을 보여주기 위하여 작은 도표편집기인 DiagramView창문부품의 코드를 제시한다. 창문부품은 2종류의 도형(직4각형칸과 직선)을 유지하며 사용자가 새로운 직4각형과 직선들을 추가하고 그것들을 복사하고 붙이기하고 삭제하며 그 속성들을 편집하게 하는 상황차림표를 제공한다.

```
class DiagramView : public QCanvasView
{
    Q_OBJECT
public:
    DiagramView(QCanvas *canvas, QWidget *parent = 0, const char *name = 0);
public slots:
    void cut();
    void copy();
    void paste();
    void del();
    void properties();
    void addBox();
    void addLine();
    void bringToFront();
    void sendToBack();
```

DiagramView클라스는 QCanvasView를 계승하며 QCanvasView는 QScrollView을 계승한다. 이 클라스는 응용프로그람이 련결할수 있는 많은 공개처리부들을 제공한다. 처리부들은 또한 창문부품이 자체로 상황차림표를 실현하는데 사용된다.

```
protected:
    void contentsContextMenuEvent(QContextMenuEvent *event);
    void contentsMousePressEvent(QMouseEvent *event);
    void contentsMouseMoveEvent(QMouseEvent *event);
    void contentsMouseDoubleClickEvent(QMouseEvent *event);
private:
    void createActions();
    void addItem(QCanvasItem *item);
    void setActiveItem(QCanvasItem *item);
    void showNewItem(QCanvasItem *item);
    QCanvasItem *pendingItem;
    QCanvasItem *activeItem;
    QPoint lastPos;
```

```
    int minZ;
    int maxZ;
    QAction *cutAct;
    QAction *copyAct;
    ...
    QAction *sendToBackAct;
};
```

클라스의 보호 및 비공개성원들을 간단히 설명한다.

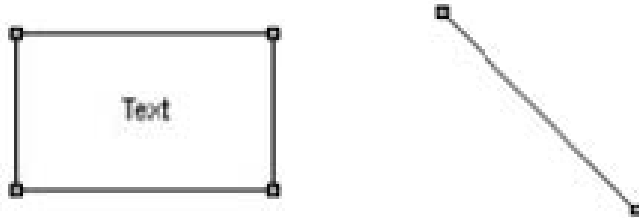



그림 8-10. DiagramBox와 DiagramLine캔버스항목

또한 DiagramView클라스중에서 2개의 사용자정의캔버스항목클라스들을 정의하여 그리려고 하는 도형들을 표시한다. 이 클라스들의 이름은 DiagramBox와 DiagramLine이다.

```
class DiagramBox : public QCanvasRectangle
{
public:
    enum { RTTI = 1001 };
    DiagramBox(QCanvas *canvas);
    ~DiagramBox();
    void setText(const QString &newText);
    QString text() const { return str; }
    void drawShape(QPainter &painter);
    QRect boundingRect() const;
    int rtti() const { return RTTI;}
private:
    QString str;
};
```

DiagramBox클라스는 직4각형과 본문의 부분을 현시하는 캔버스항목의 형이다. 이 클라스는 직4각형을 현시하는 QCanvasItem의 파생클라스 QCanvasRectangle로부터 그 기능을 계승한다. QCanvasRectangle에 직4각형의 중간에 본문을 표시하는 능력과 매개 구석에 아주 작은 바른4각형(《손잡이》)을 표시하고 항목이 능동이라는것을 가리키는 능력을 추가한다. 현실세계의 응용프로그람에서는 손잡이를 찰칵하고 끌어서 칸의 크기를 조절할수 있게 하지만 여기서

는 코드를 간단하게 만든다.

rtti()함수는 QCanvasItem로부터 재정의된다. 함수의 이름은 실행시형식별(run-time type identification)을 의미하며 그 돌림값과 RTTI상수를 비교함으로써 캔버스안의 임의의 항목이 DiagramBox인가 아닌가를 결정할수 있다. C++의 dynamic_cast<T>()기구에 의하여 같은 검사를 할수 있으나 그 기능을 유지하는 C++콤파일러들에 제한된다.

1001값은 임의로취한것이다. 1000이상의 값을 받아들일수 있다. 이 값은 같은 응용프로그람에서 사용한 다른 항목형들과 혼돈하지 말아야 한다.

```
class DiagramLine : public QCanvasLine
{
public:
    enum { RTTI = 1002 };
    DiagramLine(QCanvas *canvas);
    ~DiagramLine();
    QPoint offset() const { return QPoint((int)x(), (int)y()); }
    void drawShape(QPainter &painter);
    QPointArray areaPoints() const;
    int rtti() const { return RTTI; }
};
```

DiagramLine클라스는 직선을 표시하는 캔버스항목이다. 이 클라스는 QCanvasLine로부터 기능을 계승하며 량끝에 손잡이들을 표시하는 능력을 추가하여 직선이 능동이라는것을 가리킨다.

이제는 이 3개 클라스의 실현을 고찰한다.

```
DiagramView::DiagramView(QCanvas *canvas, QWidget *parent, const char *name)
    : QCanvasView(canvas, parent, name)
{
    pendingItem = 0;
    activeItem = 0;
    minZ = 0;
    maxZ = 0;
    createActions();
}
```

DiagramView구성자는 첫 인수로서 canvas를 가지며 그것을 기초클라스구성자에 넘긴다. DiagramView는 이 캔버스를 표시한다.

QAction들은 createActions()비공개함수에서 창조된다. 이미 앞장들에서 이 함수를 실현하였으며 여기서도 같은 형태이므로 다시 생성하지 않는다.

```
void DiagramView::contentsContextMenuEvent(QContextMenuEvent *event)
{
    QPopupMenu contextMenu(this);
    if (activeItem) {
        cutAct->addTo(&contextMenu);
        copyAct->addTo(&contextMenu);
        deleteAct->addTo(&contextMenu);
        contextMenu.insertSeparator();
        bringToFrontAct->addTo(&contextMenu);
        sendToBackAct->addTo(&contextMenu);
        contextMenu.insertSeparator();
        propertiesAct->addTo(&contextMenu);
    } else {
        pasteAct->addTo(&contextMenu);
        contextMenu.insertSeparator();
        addBoxAct->addTo(&contextMenu);
        addLineAct->addTo(&contextMenu);
    }
    contextMenu.exec(event->globalPos());
}
```

contentsContextMenuEvent()함수는 QScrollView로부터 재정의되여 상황차림표를 창조한다.

그림 8-11. DiagramView창문부품의 상황차림표들

항목이 능동이면 차림표는 항목에서 의미를 가지는 작용들 즉 Cut, Copy, Delete, Bring to Front, Send to Back, Properties들로 채워진다. 그렇지 않으면 차림표는 Paste, Add Box, Add Line.로 채워진다.

```
void DiagramView::addBox()
```

```
{
    addItem(new DiagramBox(canvas()));
}
void DiagramView::addLine()
{
    addItem(new DiagramLine(canvas()));
}
```

addBox()와 addLine()처리부들은 캔버스에 대하여 DiagramBox이나 DiagramLine항목을 창조한 다음 addItem()를 호출하여 나머지 작업을 수행한다.

```
void DiagramView::addItem(QCanvasItem *item)
{
    delete pendingItem;
    pendingItem = item;
    setActiveItem(0);
    setCursor(crossCursor);
}
```

addItem()비공개함수는 유표를 +유표로 바꾸고 pendingItem을 새로 창조된 항목으로 설정한다. 항목은 show()를 호출하기전에는 캔버스에서 보이지 않는다.

사용자가 상황차림표로부터 Add Box나 Add Line을 선택하면 유표는 +유표로 달라진다. 항목은 사용자가 캔버스우에서 찰칵할 때까지 실제로 추가되지 않는다.

```
void DiagramView::contentsMousePressEvent(QMouseEvent *event)
{
   if (event->button() == LeftButton && pendingItem) {
      pendingItem->move(event->pos().x(), event->pos().y());
      showNewItem(pendingItem);
      pendingItem = 0;
      unsetCursor();
   } else {
      QCanvasItemList items = canvas()->collisions(event->pos());
      if (items.empty())
         setActiveItem(0);
      else
         setActiveItem(*items.begin());
   }
   lastPos = event->pos();
```

}

유표가 +일 때 사용자가 왼쪽마우스단추를 찰칵하면 직4각형이나 직선을 창조할것을 요구하며 새 항목을 표시하려는 위치에서 캔버스를 찰칵한다. 항목을 찰칵한 위치에 직4각형이나 선분을 새로 표시하고 유표를 표준화살유표로 재설정한다.

캔버스에 대한 다른 마우스누르기사건은 항목을 선택하거나 선택을 해제하려는 시도로 해석된다. 캔버스에 대하여 collisions()를 호출하여 유표아래의 모든 항목들을 얻고 첫 항목을 현재 항목으로 만든다. 목록이 많은 항목을 포함하면 첫 항목은 항상 다른 항목들의 꼭대기에 그려지는 항목이다.

```
void DiagramView::contentsMouseMoveEvent(QMouseEvent *event)
{
    if (event->state() & LeftButton) {
        if (activeItem) {
            activeItem->moveBy(event->pos().x() - lastPos.x(), event->pos().y() - lastPos.y());
            lastPos = event->pos();
            canvas()->update();
        }
    }
}
```

사용자는 항목우에서 왼쪽마우스단추를 누르고 끌기하여 항목을 캔버스우에서 옮길수 있다. 마우스이동사건을 얻을 때마다 마우스가 이동한 수평 및 수직거리만큼 항목을 옮기고 캔버스에 대하여 update()를 호출한다. 캔버스항목을 수정할 때마다 update()를 호출하여 캔버스에 그 자체를 다시 그려야 한다는것을 통지한다.

```
void DiagramView::contentsMouseDoubleClickEvent(QMouseEvent *event)
{
    if (event->button() == LeftButton && activeItem &&
                            activeItem->rtti() == DiagramBox::RTTI) {
        DiagramBox *box = (DiagramBox *)activeItem;
        bool ok;
        QString newText = QInputDialog::getText(tr("Diagram"), tr("Enter new text:"),
                        QLineEdit::Normal, box->text(), &ok, this);
        if (ok) {
            box->setText(newText);
            canvas()->update();
        }
    }
```

```
}
```

사용자가 항목을 두번 찰칵하면 항목의 rtti()함수를 호출하고 그 돌림값을 DiagramBox::RTTI (1001로 정의된다)와 비교한다.

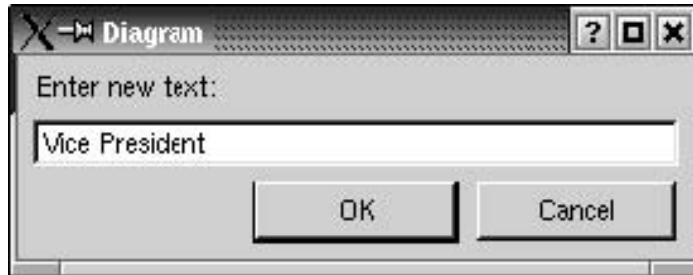



그림 8-12. DiagramBox항목의 본문변경

항목이 DiagramBox이면 QInputDialog를 펼치고 사용자가 칸에 표시되는 본문을 변경하게 한다. QInputDialog클라스는 표식자, 행편집기, *OK*단추, *Cancel*단추를 각각 하나씩 제공한다.

```
void DiagramView::bringToFront()
{
    if (activeItem) {
        ++maxZ;
        activeItem->setZ(maxZ);
        canvas()->update();
    }
}
```

bringToFront()처리부는 캔버스에서 현재 능동인 항목이 다른 항목들의 꼭대기에 놓이게 한다. 이것은 그 항목의 *z*자리표를 다른 항목들에 설정된 값보다 더 큰 값으로 설정함으로써 달성한다. 2개 항목이 같은 (*x*,*y*)위치를 차지할 때 제일 큰 *z*값을 가지는 항목을 제일 앞에 표시한다. (*z*값이 같으면 QCanvas는 항목지적자들의 값에 따라 겹쳐놓는다.)

```
void DiagramView::sendToBack()
{
    if (activeItem) {
        --minZ;
        activeItem->setZ(minZ);
        canvas()->update();
    }
}
```

sendToBack()처리부는 캔버스에서 현재 능동인 항목을 다른 모든 항목들의 뒤에 넣는다. 이것은 그 항목의 *z*자리표를 다른 항목들의 *z*값보다 작은 값으로 설정하여 수행한다.

```
void DiagramView::cut()
```

```
{
    copy();
    del();
}
```

cut()처리부는 보통과 같다.

```
void DiagramView::copy()
{
    if (activeItem) {
        QString str;
        if (activeItem->rtti() == DiagramBox::RTTI) {
            DiagramBox *box = (DiagramBox *) activeItem;
            str = QString("DiagramBox %1 %2 %3 %4 %5")
                    .arg(box->width())
                    .arg(box->height())
                    .arg(box->pen().color().name())
                    .arg(box->brush().color().name())
                    .arg(box->text());
        } else if (activeItem->rtti() == DiagramLine::RTTI) {
            DiagramLine * line = (DiagramLine *)activeItem;
            QPoint delta = line->endPoint() -line->startPoint();
            str = QString("DiagramLine %1 %2 %3")
                    .arg(delta.x())
                    .arg(delta.y())
                    .arg(line->pen().color().name());
        }
        QApplication::clipboard()->setText(str);
    }
}
```

copy()처리부는 능동항목을 문자렬로 변환하고 문자렬을 오려둠판에 복사한다.

문자렬은 항목을 재구성하는데 필요한 모든 정보를 포함한다. 례를 들면 "My Left Foot"를 포함하는 흑백색의 320×40칸은 다음의 문자렬로 표시된다.

DiagramBox 320 40 #000000 #ffffff My Left Foot

캔버스상의 항목위치를 기억할 필요는 없다. 그 항목을 덧붙일 때 놓으려는 위치에서 마우스찰칵하면 되기때문이다. 객체를 문자렬로 변환하는것은 오려둠판기능을 추가하는 방법이다.

```
void DiagramView::paste()
{
    QString str = QApplication::clipboard()->text();
    QTextIStream in(&str);
    QString tag;
    in >> tag;
    if (tag == "DiagramBox") {
      int width;
      int height;
      QString lineColor;
      QString fillColor;
      QString text;
      in >> width >> height >> lineColor >>fillColor;
      text = in.read();
      DiagramBox *box = new DiagramBox(canvas());
      box->move(20, 20);
      box->setSize(width, height);
      box->setText(text);
      box->setPen(QColor(lineColor));
      box->setBrush(QColor(fillColor));
      showNewItem(box);
    } else if (tag == "DiagramLine") {
      int deltaX;
      int deltaY;
      QString lineColor;
      in >> deltaX >> deltaY >> lineColor;
      DiagramLine *line = new DiagramLine(canvas());
      line->move(20, 20);
      line->setPoints(0, 0, deltaX, deltaY);
      line->setPen(QColor(lineColor));
      showNewItem(line);
    }
}
```

paste()처리부는 QTextIStream을 리용하여 오려둠판의 내용을 해석한다. QTextIStream은 cin과 비슷한 방법으로 공백으로 구분한 마당들과 작업한다. >>연산자를 리용하여 매개 마당을

발취하는데 DiagramBox항목의 마지막 마당은 공백을 포함할수 있으므로 발취하지 않는다. 이 마당에서는 QTextStream::read()를 리용하여 나머지 문자렬에 읽어들인다.

```
void DiagramView::del()
{
    if (activeItem) {
        QCanvasItem *item = activeItem;
        setActiveItem(0);
        delete item;
        canvas()->update();
    }
}
```

del()처리부는 능동항목을 삭제하고 QCanvas::update()를 호출하여 캔버스를 다시 그린다.

```
void DiagramView::properties()
{
    if (activeItem) {
        PropertiesDialog dialog;
        dialog.exec(activeItem);
    }
}
```

properties()처리부는 능동항목에 대하여 Properties대화칸을 펼친다. PropertiesDialog클라스는 《고급한》 대화칸이며 단순히 대화칸에 능동항목의 지적자를 넘기면 된다.

그림 8-13. Properties대화칸의 두 형태

```
void DiagramView::showNewItem(QCanvasItem *item)
{
    setActiveItem(item);
    bringToFront();
    item->show();
    canvas()->update();
}
```

showNewItem()비공개함수는 코드안의 여러곳에서 호출되며 새로 창조한 캔버스항목을 표시하고 능동으로 한다.

```
void DiagramView::setActiveItem(QCanvasItem *item)
{
    if (item != activeItem) {
        if (activeItem)
            activeItem->setActive(false);
        activeItem = item;
        if (activeItem)
            activeItem->setActive(true);
        canvas()->update();
    }
}
```

끝으로 setActiveItem()비공개함수는 낡은 능동항목의 《능동》기발을 지우고 activeItem변수를 설정하며 새 능동항목의 기발을 설정한다. 항목의 《능동》기발은 QCanvasItem에 보관된다. Qt는 기발자체를 사용하지 않는다. 기발은 순수 파생클라스들의 편리를 위하여 제공된다. DiagramBox와 DiagramLine파생클라스들이 능동인가 아닌가에 따라서 각이하게 그것들을 그리려고 하므로 그 파생클라스들에서 기발을 리용한다.

그러면 DiagramBox와 DiagramLine의 코드를 고찰하자.

```
const int Margin = 2;
void drawActiveHandle(QPainter &painter, const QPoint &center)
{
    painter.setPen(Qt::black);
    painter.setBrush(Qt::gray);
    painter.drawRect(center.x() - Margin, center.y() - Margin, 2 * Margin + 1,
                     2 * Margin + 1);
}
```

drawActiveHandle()함수를 2개의 DiagramBox와 DiagramLine에서 사용하며 항목이 능동항목

이라는것을 가리키는 아주 작은 바른4각형을 그린다.

```
DiagramBox::DiagramBox(QCanvas *canvas) : QCanvasRectangle (canvas)
{
    setSize(100, 60);
    setPen(black);
    setBrush(white);
    str = "Text";
}
```

DiagramBox구성자에서는 직4각형의 크기를 100×60으로 설정한다. 또한 펜색을 흑색으로, 솔색을 백색으로 설정한다. 펜색은 4각형의 테두리와 본문을 그리는데 쓰이고 솔색은 4각형의 배경을 칠하는데 쓰인다.

```
DiagramBox::~DiagramBox()
{
    hide();
}
```

DiagramBox해체자는 항목에 대하여 hide()를 호출한다. 이것은 QCanvasPolygonalItem이 작업하는 방식으로 인하여 QCanvasPolygonlItem(QCanvasRectangle의 기초클라스)로부터 계승되는 모든 클라스들에 필요하다.

```
void DiagramBox::setText(const QString &newText)
{
    str = newText;
    update();
}
```

setText()함수는 직4각형안에 표시할 본문을 설정하고 QCanvasItem::update()를 호출하여 이 항목을 변경된것으로 표식한다.

```
void DiagramBox::drawShape(QPainter &painter)
{
    QCanvasRectangle::drawShape(painter);
    painter.drawText(rect(), AlignCenter, text());
    if (isActive()) {
        drawActiveHandle(painter, rect().topLeft());
        drawActiveHandle(painter, rect().topRight());
        drawActiveHandle(painter, rect().bottomLeft());
        drawActiveHandle(painter, rect().bottomRight());
    }
```

```
}
```

drawShape()함수는 본문항목이 능동인 경우에는 4개의 손잡이를 그리도록 QCanvasPolygonalItem로부터 재정의된다. 기초클라스를 리용하여 직4각형 자체를 그린다.

```
QRect DiagramBox::boundingRect() const
{
   return QRect((int)x() - Margin, (int)y() - Margin, width() + 2 * Margin,
               height() + 2 * Margin);
}
```

boundingRect()함수는 QCanvasItem으로부터 재정의된다. QCanvas에서는 이 함수를 충돌탐색과 그리기최적화에 리용한다. 함수가 돌려주는 직4각형은 적어도 drawShape()에서 그리는 령역만큼 커야 한다.

QCanvasRectangle의 기정실현은 충분하지 못하다. 왜냐하면 항목이 능동으로 되면 직4각형의 각 모서리에 그리는 손잡이들을 고려하지 않았기때문이다.

```
DiagramLine::DiagramLine(QCanvas *canvas) : QCanvasLine(canvas)
{
    setPoints(0, 0, 0, 99);
}
```

DiagramLine구성자에서는 선분을 정의하는 두점을 (0, 0)과 (0, 99)로 설정한다. 결과는 100화소길이의 수직선이다.

```
DiagramLine::~DiagramLine()
{
    hide();
}
```

다시 해체자에서 hide()를 호출해야 한다.

```
void DiagramLine::drawShape(QPainter &painter)
{
    QCanvasLine::drawShape(painter);
    if (isActive()) {
        drawActiveHandle(painter, startPoint() + offset());
        drawActiveHandle(painter, endPoint() + offset());
    }
}
```

항목이 능동이면 직선의 량끝에 손잡이를 그리도록 QCanvasLine으로부터 drawShape()함수를 재정의한다. 기초클라스를 리용하여 직선자체를 그린다. offset()함수는 DiagramLine클라스정의에서 실현된다. 이 함수는 캔버스에서 항목의 위치를 돌려준다.

```
QPointArray DiagramLine::areaPoints() const
{
    const int Extra = Margin + 1;
    QPointArray points(6);
    QPoint pointA = startPoint() + offset();
    QPoint pointB = endPoint() + offset();
    if (pointA.x() > pointB.x())
        swap(pointA, pointB);
    points[0] = pointA + QPoint(-Extra, -Extra);
    points[1] = pointA + QPoint(-Extra, +Extra);
    points[3] = pointB + QPoint(+Extra, +Extra);
    points[4] = pointB + QPoint(+Extra, -Extra);
    if (pointA.y() > pointB.y()) {
        points[2] = pointA + QPoint(+Extra, +Extra);
        points[5] = pointB + QPoint(-Extra, -Extra);
    } else {
        points[2] = pointB + QPoint(-Extra, +Extra);
        points[5] = pointA + QPoint(+Extra, -Extra);
    }
    return points;
}
```

areaPoints()함수는 DiagramBox에서 boundingRect()함수와 비슷한 역할을 한다.

경계직4각형은 대각선과 실제로 대부분의 다각형들에서 거의 비슷하다. 이것들에 대하여 areaPoints()를 재정의하고 항목이 그려지는 구역의 테두리를 돌려준다. QCanvasLine실현은 이미 직선의 아주 좋은 륜곽선을 돌려주지만 손잡이들을 고려하지 않는다.

처음으로 할 일은 두점을 pointA와 pointB에 보관하고 pointA가 pointB의 왼쪽에 놓이도록 한다. (필요하다면 swap()(<algorithm>에서 정의)로 두 점을 서로 교체한다). 그때 고려해야 할 두가지 경우가 있다. 즉 올리경사선과 내리경사선인 경우이다.

직선의 경계구역은 늘 6개의 점들로 표시되지만 이 점들은 직선이 내리경사인가 올리경사인가에 따라 달라진다. 그럼에도 불구하고 6점중 4개(번호 0, 1, 3, 4)는 두 경우에 같다. 례를 들면 점 0과 1은 늘 손잡이 A의 왼쪽웃구석과 왼쪽아래구석에 배치되지만 이와 대조적으로 점2는 올리경사선인 경우에는 손잡이 A의 오른쪽아래구석에 배치되고 내리경사선인 경우에는 손잡이 B의 왼쪽아래구석에 배치된다.

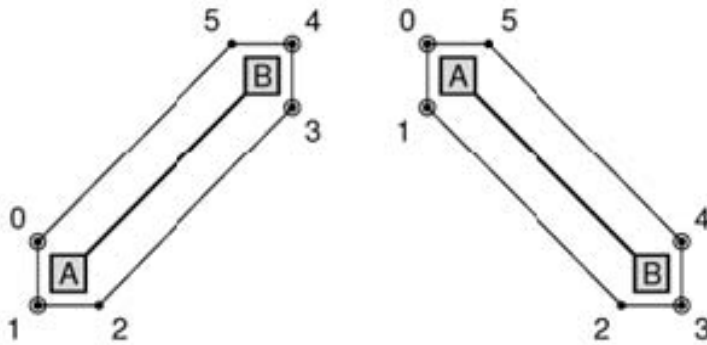



그림 8-14. DiagramLine의 경계구역

작성한 얼마 안되는 코드를 고려하면 DiagramView창문부품은 이미 상당한 기능을 제공하며 항목들의 선택 및 이동과 상황차림표를 가지고있다.

놓친것은 항목이 능동일 때 표시된 손잡이들을 항목의 크기를 변경하기 위하여 끌기할수 없는것이다. 그것을 변경하려고 한다면 여기서 사용한것과는 다른 수법을 리용해야 한다. 항목의 drawShape()함수들에서 손잡이를 그리지 않고 매개 손잡이를 캔버스항목으로 만든다. 손잡이우로 이동할 때 유표를 변경하려고 한다면 실제로 이동되는 시각에 setCursor()를 호출할 수 있다. 그러자면 마우스단추를 찰칵할 때 Qt는 보통 마우스이동사건들만 송신하므로 우선 setMouseTracking(true)를 호출해야 한다.

또한 다중선택과 항목그룹화를 유지하도록 개량할수 있다.

이 절에서는 QCanvas와 QCanvasView사용의 동작실례를 제공하였으나 QCanvas의 기능을 모두 설명하지는 않았다. 례를 들면 캔버스항목들은 setVelocity()를 호출하여 규칙적인 시격으로 캔버스우를 이동함으로써 설정할수 있다.(자세한 내용은 QCanvas와 그 관련클라스들의 문서를 보시오.)

제3절. 인쇄

Qt에서 인쇄는 창문부품이나 픽스매프상에서 그리기와 비슷하다. 인쇄는 다음의 단계들로 이루어진다.

① 그리기장치로서 작업하는 QPrinter를 창조한다.

② QPrinter::setup()를 호출하여 인쇄대화칸을 열고 사용자가 인쇄기를 선택하고 몇가지 항목을 설정하게 한다.

③ QPrinter에 조작하는 QPainter를 창조한다.

④ QPainter에 의해 페지를 그린다.

⑤ QPrinter::newPage()를 호출하여 다음 페지로 넘어간다.

⑥ 걸음 ④와 ⑤를 모든 페지를 인쇄할 때까지 반복한다.

Windows와 Mac OS X에서 QPrinter는 체계의 인쇄기구동프로그람을 리용한다. Unix에서 QPrinter는 PostScript를 생성하고 그것을 lp 혹은 lpr(QPrinter::setPrintProgram()에 의해 설정된 프로그람)에로 송신한다.

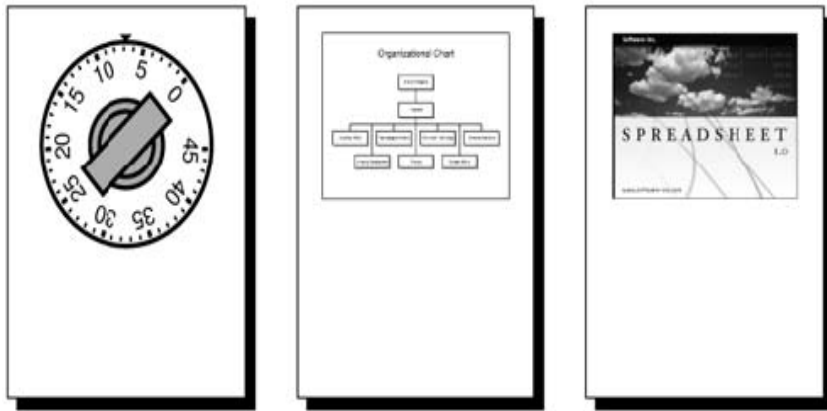



그림 8-15. OverTimer, QCanvas, QImager의 인쇄

한 페지우에 모두 인쇄하는 간단한 실례부터 보기로 하자. 첫 실례는 OvenTimer창문부품을 인쇄한다.

```
void PrintWindow::printOvenTimer(OvenTimer *ovenTimer)
{
    if (printer.setup(this)) {
        QPainter painter(&printer);
        QRect rect = painter.viewport();
        int side = QMIN(rect.width(), rect.height());
        painter.setViewport(0, 0, side, side);
        painter.setWindow(-50, -50, 100, 100);
        ovenTimer->draw(&painter);
    }
}
```

PrintWindow클라스가 QPrinter형의 printer라는 성원변수를 가진다고 가정한다. 간단히 printOvenTimer()에서 탄창에 QPrinter를 창조할수 있으나 한 인쇄에서 다른 인쇄에로 사용자의 설정을 넘길수 없다.

setup()를 호출하여 인쇄대화칸을 펼친다. 대화칸은 사용자가 OK단추를 찰칵하면 true를 돌려주고 그렇지 않으면 false를 돌려준다. setup()호출후에 QPrinter객체는 사용준비가 되여있다.

QPainter를 창조하여 QPrinter에 그리기한다. 그다음 painter의 보기구역바른4각형을 만들고 painter의 창문을 OvenTimer가 요구하는 직4각형인 (-50, -50, 100, 100)으로 초기화한다. draw()를 호출하여 그리기를 수행한다. 보기구역바른4각형을 설정하지 않으면 OvenTimer는 페지의 전체높이만큼 수직으로 늘어난다.

기정으로 QPainter의 창문은 printer가 화면과 비슷한 분해능(보통 72～100dpi)을 가지도록 초기화되고 이것은 인쇄에서 창문부품그리기코드를 간단히 재리용하게 한다. 여기서는 자체

의 창문을 (-50, -50, 100, 100)으로 설정하였으므로 문제가 없다.

OvenTimer인쇄는 창문부품이 화면상에서 사용자와의 교제를 위하여 예정된것이므로 그리 현실적인 실례가 못된다. 그러나 5장에서 개발한 Plotter창문부품과 같은 다른 창문부품들에서는 창문부품의 그리기코드를 인쇄에 재리용하는것이 큰 의의가 있다.

더 실천적인 실례는 QCanvas인쇄이다. 캔버스를 사용하는 응용프로그람들은 흔히 사용자가 그린것을 인쇄할수 있어야 한다. 이것은 다음과 같이 일반적인 방법으로 수행할수 있다.

```
void PrintWindow::printCanvas(QCanvas *canvas)
{
    if (printer.setup(this)) {
        QPainter painter(&printer);
        QRect rect = painter.viewport();
        QSize size = canvas->size();
        size.scale(rect.size(), QSize::ScaleMin);
        painter.setViewport(rect.x(), rect.y(), size.width(), size.height());
        painter.setWindow(canvas->rect());
        painter.drawRect(painter.window());
        painter.setClipRect(painter.viewport());
        QCanvasItemList items = canvas->collisions(canvas->rect());
        QCanvasItemList::const_iterator it = items.end();
        while (it != items.begin()) {
            --it;
            (*it)->draw(painter);
        }
    }
}
```

이번에는 painter의 창문을 캔버스의 경계직4각형으로 설정하고 같은 가로세로비를 가지는 직4각형으로 보기구역을 제한한다. 이것을 달성하려면 둘째인수로서 ScaleMin를 가지는 QSize::scale()를 사용한다. 례를 들면 캔버스가 640×480크기를 가지고 painter의 보기구역크기가 5000×5000이면 우리가 사용하는 결과의 보기구역크기는 5000×3750이다.

인수로서 캔버스의 직4각형을 넘기여 collisions()를 호출하여 제일 큰것부터 제일 작은 *z*값에 따라 정렬된 볼수 있는 모든 캔버스항목들의 목록을 얻는다. 보다 큰 *z*값을 가지는 항목들보다 먼저 보다 작은 *z*값을 가지는 항목들을 그리기 위하여 목록을 끝으로부터 순환하면서 그것들에 대하여 QCanvasItem::draw()를 호출한다. 이것은 앞에 있는 항목들이 뒤에 있는 항목들의 꼭대기에 그려진다는것을 담보한다.

셋째 실례는 QImage를 그리는것이다.

```
void PrintWindow::printImage(const QImage &image)
{
    if (printer.setup(this)) {
        QPainter painter(&printer);
        QRect rect = painter.viewport();
        QSize size = image.size();
        size.scale(rect.size(), QSize::ScaleMin);
        painter.setViewport(rect.x(), rect.y(),size.width(), size.height());
        painter.setWindow(image.rect());
        painter.drawImage(0, 0, image);
    }
}
```

창문을 화상의 직4각형으로 설정하고 보기구역을 같은 가로세로비를 가지는 직4각형으로 설정하고 위치 (0, 0)에 화상을 그린다.

한페지밖에 안되는 항목들의 인쇄는 간단하다. 그러나 많은 응용프로그람들은 여러 페지에 인쇄해야 한다. 그러한 경우에 한번에 1페지를 그리고 newPage()를 호출하여 다음 페지로 넘어가야 한다. 이것은 각 페지에 인쇄할수 있는 정보가 얼마인가를 결정하는 문제를 제기한다.

Qt에서 여러페지문서를 처리하는 두가지 방법이 있다.

· 요구하는 자료를 HTML로 변환하고 Qt의 리치본문엔진 QSimpleRichText를 사용하여 그것을 표시한다.

· 손에 의한 그리기와 페지분할을 수행한다.

두가지 수법을 차례로 고찰하자.

실례로 본문으로 서술한 꽃이름들의 목록인 꽃편람을 인쇄하려고 한다. 편람에서 매개 항목은 《이름:설명》형식의 문자렬로 보관된다. 례를 들면

Miltonopsis santanae: 가장 위험한 란초과.

매개 꽃의 자료가 단일문자렬로 표시되므로 하나의 QStringList에 의해 편람안의 꽃들을 모두 표시할수 있다.

여기에 Qt의 리치본문엔진에 의해 꽃편람을 인쇄하는 함수가 있다.

```
void PrintWindow::printFlowerGuide(const QStringList &entries)
{
    QString str;
    QStringList::const_iterator it = entries.begin();
    while (it != entries.end()) {
        QStringList fields = QStringList::split(": ", *it);
```

```
        QString title = QStyleSheet::escape(fields[0]);
        QString body = QStyleSheet::escape(fields[1]);
        str += "<table width=\"100%\" border=1 cellspacing=0>\n"
               "<tr><td bgcolor=\"lightgray\"><font size=\"+1\">"
               "<b><i>" + title + "</i></b></font>\n<tr><td>" + body + "\n</table>\n<br>\n";
        ++it;
    }
    printRichText(str);
}
```

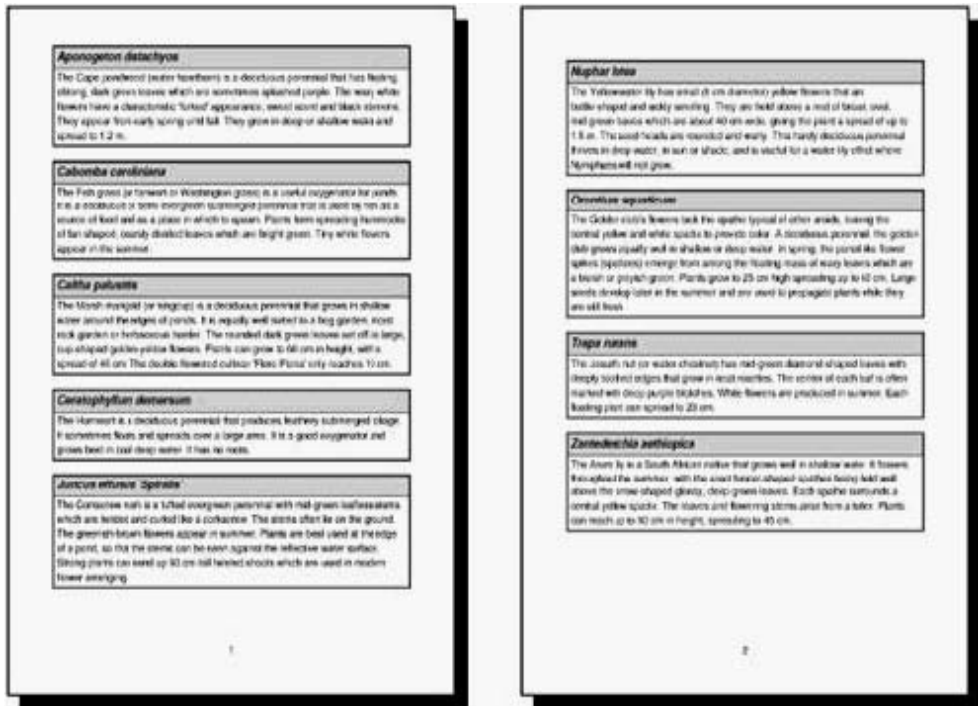



그림 8-16. QSimpleRichText에 의한 꽃편람의 인쇄

첫 단계는 자료를 HTML로 변환하는것이다. 매개 꽃은 2개 세포를 가지는 HTML표로 된다. QStyleSheet::escape()에 의하여 특수문자 &, <, >를 대응하는 HTML항목들("&", "<", ">")로 교체한다. 그다음 printRichText()를 호출하여 본문을 인쇄한다.

```
const int LargeGap = 48;
void PrintWindow::printRichText(const QString &str)
{
    if (printer.setup(this)) {
        QPainter painter(&printer);
        int pageHeight = painter.window().height() -2 * LargeGap;
        QSimpleRichText richText(str, bodyFont, "", 0, 0, pageHeight);
```

```
            richText.setWidth(&painter, painter.window().width());
            int numPages = (int)ceil((double)richText.height() / pageHeight);
            int index;
            for (int i = 0; i < (int)printer.numCopies(); ++i) {
              for (int j = 0; j < numPages; ++j) {
                  if (i > 0 || j > 0)
                      printer.newPage();
                  if (printer.pageOrder() == QPrinter::LastPageFirst)
                  {
                      index = numPages -j -1;
                  } else {
                      index = j;
                  }
                  printPage(&painter, richText, pageHeight, index);
              }
            }
        }
    }
```

printRichText()함수는 HTML문서의 인쇄를 고려한다. 이 함수는 임의의 Qt응용프로그람에서 임의의 HTML을 인쇄하는데 리용할수 있다.

창문크기와 머리부(header)와 꼬리부(footer)용으로 폐지의 꼭대기와 밑에 남기려는 짬의 크기를 고려하여 1폐지의 높이를 계산한다. 그다음 HTML자료를 포함하는 QSimpleRichText객체를 창조한다. QSimpleRichText구성자의 마지막 인수는 폐지높이이고 QSimpleRichText는 그것을 사용하여 마음에 드는 폐지가르기를 생성한다.

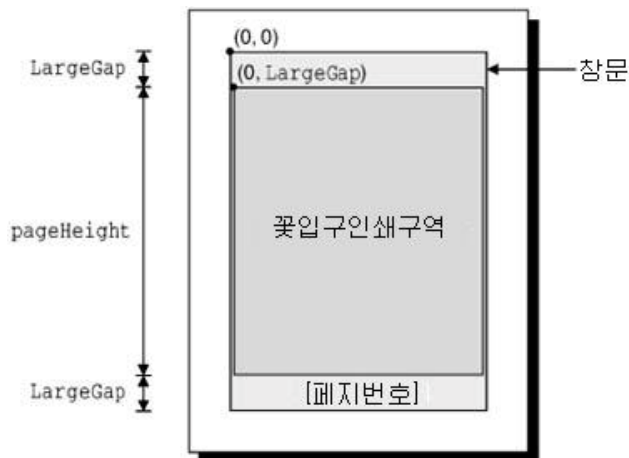



그림 8-17. 꽃편람의 폐지배치

그다음 각 페지를 인쇄한다. 사용자가 요구한 사본수를 생성하기 위하여 필요한 회수만큼 바깥쪽 for순환을 반복한다. 대부분의 인쇄기구동프로그람들은 여러 사본을 유지하므로 이것들에 대하여 QPrinter::numCopies()는 늘 1을 돌려준다. 인쇄기구동프로그람이 여러 사본을 유지하지 않으면 numCopies()는 사용자가 요구하는 사본수를 돌려주며 응용프로그람은 그 량만큼 인쇄하는데 응답할수 있다. 앞의 실례들에서는 단순성을 위하여 numCopies()를 무시하였다.

안쪽 for순환은 페지들을 순환한다. 첫 페지가 아니면 newPage()를 호출하여 낡은 페지를 지우고 새 페지에 대한 그리기를 시작한다. printPage()를 호출하여 매개 페지를 그린다.

인쇄대화칸은 사용자가 반대순서로 페지들을 인쇄하게 한다.

printer, bodyFont, footerFont가 PrintWindow클라스의 성원변수라고 가정한다.

```
void PrintWindow::printPage(QPainter *painter, const QSimpleRichText &richText,
                            int pageHeight, int index)
{
    QRect rect(0, index * pageHeight + LargeGap, richText.width(), pageHeight);
    painter->saveWorldMatrix();
    painter->translate(0, -rect.y());
    richText.draw(painter, 0, LargeGap, rect, colorGroup());
    painter->restoreWorldMatrix();
    painter->setFont(footerFont);
    painter->drawText(painter->window(), AlignHCenter | AlignBottom,
              QString::number(index + 1));
}
```

printPage()함수는 문서의 (index+1)번째 페지를 인쇄한다. 페지는 HTML과 꼬리부구역안의 페지번호로 이루어진다.

QPainter를 해석하고 위치와 그리려는 리치본문의 몫을 지정하는 직4각형을 인수로 하여 draw()를 호출한다. 이것은 높이가 pageHeight인 작은 부분들로 갈라야 하는 아주 긴 하나의 페지로서 리치본문을 보여주도록 한다.

그다음 페지바닥의 중심에 페지번호를 그린다. 매 페지우에 머리부를 주려면 여분의 drawText()호출을 사용해야 한다.

LargeGap상수는 48로 설정된다. 96dpi의 화면분해능을 가정하면 이것은 0.5in(12.7mm)이다. 화면분해능에 관계없이 정확한 길이를 얻으려면 다음과 같이 QPaintDeviceMetrics클라스를 사용해야 한다.

```
QPaintDeviceMetrics metrics(&printer);
int LargeGap = metrics.logicalDpiY() / 2;
```

여기에 PrintWindow구성자에서 bodyFont와 footerFont를 초기화하는 방법이 있다.

```
bodyFont = QFont("Helvetica", 14);
footerFont = bodyFont;
```

그러면 QPainter에 의해 꽃편람을 그리는 방법을 보기로 하자. 여기에 새로운 printFlowerGuide()함수가 있다.

```
void PrintWindow::printFlowerGuide(const QStringList &entries)
{
    if (printer.setup(this)) {
        QPainter painter(&printer);
        vector<QStringList> pages;
        int index;
        paginate(&painter, &pages, entries);
        for (int i = 0; i < (int)printer.numCopies(); ++i) {
            for (int j = 0;j < (int)pages.size();++j) {
                if (i > 0 || j > 0)
                    printer.newPage();
                if (printer.pageOrder() == QPrinter::LastPageFirst) {
                    index = pages.size() - j - 1;
                } else {
                    index = j;
                }
                printPage(&painter, pages, index);
            }
        }
    }
}
```

인쇄기설정과 painter구성후에 처음으로 할 일은 paginate()보조함수를 호출하여 어느 항목이 어느 페지에 나타나야 하는가를 결정하는것이다. 그 결과는 QStringList들의 벡토르로서 매개 QStringList가 한 페지에 대한 항목들을 보관한다.

례를 들면 꽃편람이 6개의 항목을 포함하고 그것을 A, B, C, D, E, F로서 참고한다고 가정하자. 이제 첫 페지에 A와 B의 자리가 있고 둘째 페지에 C, D, E, 셋째 페지에 F의 자리가 있다고 가정하자. 그때 페지벡토르는 첨수위치 0에 목록 [A, B], 첨수위치1에 목록 [C, D, E], 첨수위치2에 목록 [F]를 가질수 있다.

함수의 나머지는 printRichText()에서 처음에 수행한것과 거의 같다. 그러나 printPage()함수는 간단히 알수 있겠지만 다르다.

```
void PrintWindow::paginate(QPainter *painter, vector<QStringList> *pages,
```

```
                    const QStringList &entries)
{
    QStringList currentPage;
    int pageHeight = painter->window().height() - 2 * LargeGap;
    int y = 0;
    QStringList::const_iterator it = entries.begin();
    while (it != entries.end()) {
        int height = entryHeight(painter, *it);
        if (y + height > pageHeight && !currentPage.empty()) {
            pages->push_back(currentPage);
            currentPage.clear();
            y = 0;
        }
        currentPage.push_back(*it);
        y += height + MediumGap;
        ++it;
    }
    if (!currentPage.empty())
        pages->push_back(currentPage);
}
```

paginate()함수는 꽃편람항목들을 페지들에 분배한다. 이것은 한 항목의 높이를 계산하는 entryHeight()함수에 기초하고있다.

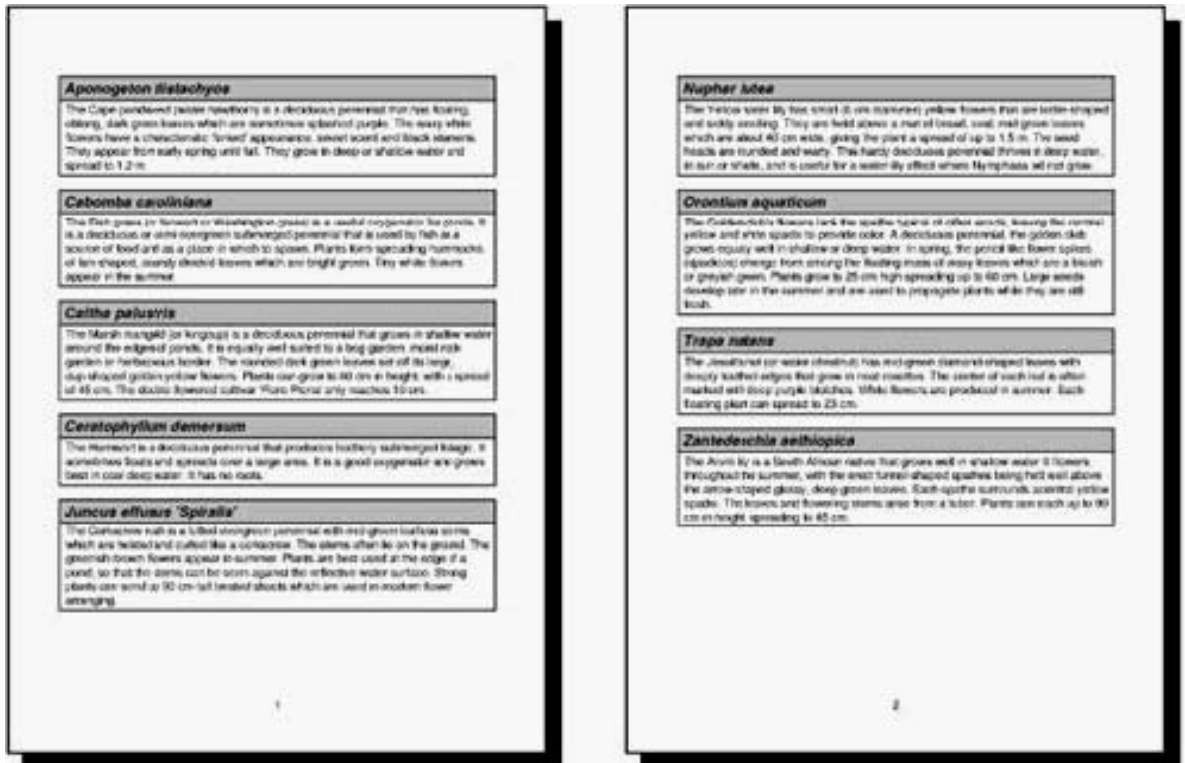



그림 8-18. QPainter에 의한 꽃편람의 인쇄

항목들을 순환하면서 적합하지 않은 항목에 올 때까지 현재 페지에 그것들을 추가한 다음 현재 페지를 페지벡토르에 추가하고 새 페지를 시작한다.

```
int PrintWindow::entryHeight(QPainter *painter, const QString &entry)
{
    QStringList fields = QStringList::split(": ", entry);
    QString title = fields[0];
    QString body = fields[1];
    int textWidth = painter->window().width() - 2 * SmallGap;
    int maxHeight = painter->window().height();
    painter->setFont(titleFont);
    QRect titleRect = painter->boundingRect(0, 0, textWidth, maxHeight, WordBreak, title);
    painter->setFont(bodyFont);
    QRect bodyRect = painter->boundingRect(0, 0, textWidth, maxHeight, WordBreak, body);
    return titleRect.height() + body Rect.height() + 4 * SmallGap;
}
```

entryHeight()함수는 QPainter::boundingRect()에 의하여 한개 항목에 필요한 수직공간을 계산한다. 그림 8-19는 꽃항목의 배치와 SmallGap와 MediumGap상수들의 의미를 보여준다.

SmallGap SmallGap SmallGap
Title
SmallGap
SmallGap
Body
SmallGap
MediumGap

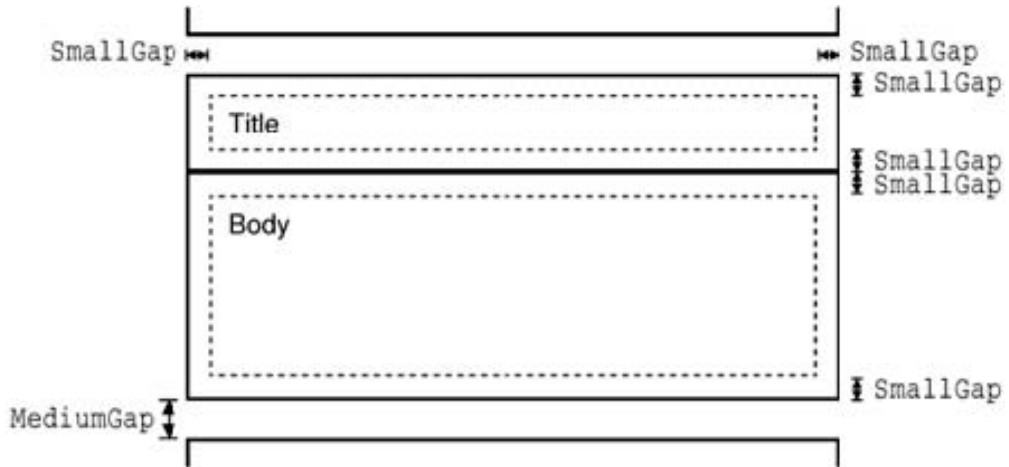



그림 8-19. 꽂항목의 배치

```
void PrintWindow::printPage(QPainter *painter, const vector<QStringList> &pages, int index)
{
    painter->saveWorldMatrix();
    painter->translate(0, LargeGap);
    QStringList::const_iterator it = pages[index].begin();
    while (it != pages[index].end()) {
        QStringList fields = QStringList::split(": ", *it);
        QString title = fields[0];
        QString body = fields[1];
        printBox(painter, titleFont, title, lightGray);
        printBox(painter, bodyFont, body, white);
        painter->translate(0, MediumGap);
        ++it;
    }
    painter->restoreWorldMatrix();
    painter->setFont(footerFont);
    painter->drawText(painter->window(), AlignHCenter | AlignBottom,
                QString::number(index + 1));
}
```

printPage()함수는 꽂편람항목들을 모두 순환하면서 printBox()를 두번 호출하여 제목(꽂이름)과 본체(설명)를 출력한다. 또한 이때 페지바닥의 중심에 페지번호를 그린다.

```
void PrintWindow::printBox(QPainter *painter, const QFont &font, const QString &str,
                            const QBrush &brush)
{
```

```
        painter->setFont(font);
        int boxWidth = painter->window().width();
        int textWidth = boxWidth -2 * SmallGap;
        int maxHeight = painter->window().height();
        QRect textRect = painter->boundingRect(SmallGap, SmallGap, textWidth, maxHeight,
                                WordBreak, str);
        int boxHeight = textRect.height() + 2 * SmallGap;
        painter->setPen(QPen(black, 2, SolidLine));
        painter->setBrush(brush);
        painter->drawRect(0, 0, boxWidth, boxHeight);
        painter->drawText(textRect, WordBreak, str);
        painter->translate(0, boxHeight);
    }
```

printBox()함수는 직4각형의 틀선을 그린 다음 칸에 본문을 그린다.

사용자가 긴 문서를 인쇄하거나 짧은 문서를 여러개 요구한다면 QProgressDialog를 펼치여 사용자에게 (Cancel을 찰칵하여) 인쇄조작을 취소할 기회를 주는것이 보통 좋다. 여기에 이것을 수행하는 printFlowerGuide()의 수정판이 있다.

```
    void PrintWindow::printFlowerGuide(const QStringList &entries)
    {
        if (printer.setup(this)) {
            QPainter painter(&printer);
            vector<QStringList> pages;
            int index;
            paginate(&painter, &pages, entries);
            int numSteps = printer.numCopies() * pages.size();
            int step = 0;
            QProgressDialog progress(tr("Printing file..."), tr("Cancel"), numSteps, this);
            progress.setModal(true);
            for (int i = 0;i < (int)printer.numCopies();++i) {
                for (int j = 0;j < (int)pages.size();++j) {
                    progress.setProgress(step);
                    qApp->processEvents();
                    if (progress.wasCanceled()) {
                        printer.abort();
                        return;
```

```
                }
                ++step;
                if (i > 0 || j > 0)
                    printer.newPage();
                if (printer.pageOrder() == QPrinter::LastPageFirst) {
                    index = pages.size() - j - 1;
                } else {
                    index = j;
                }
                printPage(&painter, pages, index);
            }
        }
    }
}
```

사용자가 Cancel을 찰칵할 때 QPrinter::abort()를 호출하여 인쇄조작을 중지한다.

제4절. OpenGL에 의한 도형처리

OpenGL은 2차원과 3차원 도형처리를 위한 표준API이다. Qt응용프로그람들은 Qt의 QGL모듈을 리용하여 OpenGL도형을 그릴수 있다. 이 절에서는 독자가 OpenGL을 알고있다고 가정한다.

Qt응용프로그람으로부터 OpenGL에 의한 도형처리는 간단하다. QGLWidget의 파생클라스를 만들고 가상함수들을 재정의하고 응용프로그람을 QGL과 OpenGL서고들에 련결하여야 한다. QGLWidget가 QWidget로부터 계승되므로 우리가 이미 알고있는것의 대부분을 여전히 적용할수 있다. 주요한 차이는 표준OpenGL함수들을 사용하여 QPainter대신에 그리기를 수행한다는것이다.

그 작업방법을 보여주기 위하여 그림 8-20에 보여주는 Cube응용프로그람의 코드를 개괄한다. 응용프로그람은 서로 다른 색의 면을 가지는 3차원 정6면체를 표시한다. 사용자는 마우스단추를 누르고 끌기하여 립방체를 회전시킬수 있다. 사용자는 립방체의 한 면을 두번 찰칵하여 펼쳐지는 QColorDialog로부터 색을 선택하여 면의 색을 설정할수 있다.

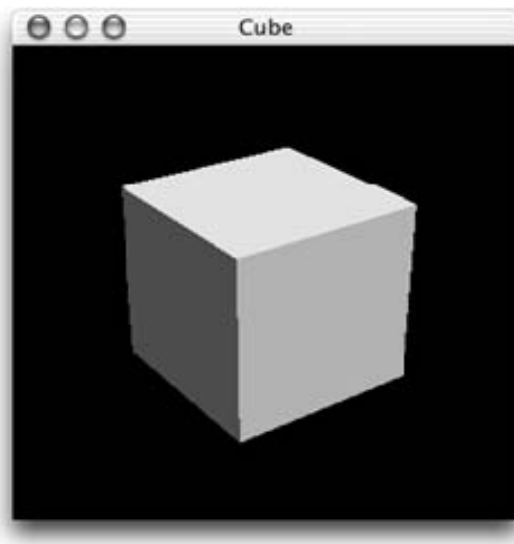



그림 8-20. Cube응용프로그람

```
class Cube : public QGLWidget
{
public:
    Cube(QWidget *parent = 0, const char *name = 0);
protected:
    void initializeGL();
    void resizeGL(int width, int height);
    void paintGL();
    void mousePressEvent(QMouseEvent *event);
    void mouseMoveEvent(QMouseEvent *event);
    void mouseDoubleClickEvent(QMouseEvent *event);
private:
    void draw();
    int faceAtPosition(const QPoint &pos);
    GLfloat rotationX;
    GLfloat rotationY;
    GLfloat rotationZ;
    QColor faceColors[6];
    QPoint lastPos;
};
```

Cube는 QGLWidget를 계승한다. initializeGL(), resizeGL(), paintGL()함수들은 QGLWidget로부터 재정의된다. 마우스사건처리함수들은 보통과 같이 QWidget로부터 재정의된다. QGLWidget

는 <qgl.h>에서 정의된다.

```
Cube::Cube(QWidget *parent, const char *name) : QGLWidget(parent, name)
{
    setFormat(QGLFormat(DoubleBuffer | DepthBuffer));
    rotationX = 0;
    rotationY = 0;
    rotationZ = 0;
    faceColors[0] = red;
    faceColors[1] = green;
    faceColors[2] = blue;
    faceColors[3] = cyan;
    faceColors[4] = yellow;
    faceColors[5] = magenta;
}
```

구성자에서는 QGLWidget::setFormat()를 호출하여 OpenGL현시상황을 지정하고 클라스의 비공개변수들을 초기화한다.

```
void Cube::initializeGL()
{
    qglClearColor(black);
    glShadeModel(GL_FLAT);
    glEnable(GL_DEPTH_TEST);
    glEnable(GL_CULL_FACE);
}
```

initializeGL()함수는 paintGL()이 호출되기전에 한번 호출된다. 이것은 OpenGL묘사상황을 설정하고 현시목록을 정의하며 다른 초기화를 수행한다.

모든 코드는 QGLWidget의 qglClearColor()함수호출을 제외하고는 표준 OpenGL이다. 표준 OpenGL로 고착시키려고 한다면 RGBA방식에서는 glClearColor()를, 색첨수방식에서는 glClearIndex()를 각각 대신에 호출한다.

```
void Cube::resizeGL(int width, int height)
{
    glViewport(0, 0, width, height);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    GLfloat x = (GLfloat)width / height;
    glFrustum(-x, x, -1.0, 1.0, 4.0, 15.0);
```

```
    glMatrixMode(GL_MODELVIEW);
}
```

resizeGL()함수는 paintGL()이 처음으로 호출되기전에, 그러나 initializeGL()이 호출된 후에 한번 호출된다. 이것은 창문부품의 크기에 따라 OpenGL보기구역, 사영, 다른 환경을 설정할 수 있는 위치이다.

```
void Cube::paintGL()
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    draw();
}
```

paintGL()함수는 창문부품을 다시 그려야 할 때마다 호출된다. 이것은 QWidget::paintEvent()와 비슷하지만 QPainter함수들대신에 OpenGL함수들을 사용한다. 실제의 그리기는 비공개함수draw()에 의해 수행된다.

```
void Cube::draw()
{
    static const GLfloat coords[6] [4] [3] = {
    { { +1.0, -1.0, +1.0 }, { +1.0, -1.0, -1.0 }, { +1.0, +1.0, -1.0 }, { +1.0, +1.0, +1.0 } },
    { { -1.0, -1.0, -1.0 }, { -1.0, -1.0, +1.0 }, { -1.0, +1.0, +1.0 }, { -1.0, +1.0, -1.0 } },
    { { +1.0, -1.0, -1.0 }, { -1.0, -1.0, -1.0 }, { -1.0, +1.0, -1.0 }, { +1.0, +1.0, -1.0 } },
    { { -1.0, -1.0, +1.0 }, { +1.0, -1.0, +1.0 }, { +1.0, +1.0, +1.0 }, { -1.0, +1.0, +1.0 } },
    { { -1.0, -1.0, -1.0 }, { +1.0, -1.0, -1.0 }, { +1.0, -1.0, +1.0 }, { -1.0, -1.0, +1.0 } },
    { { -1.0, +1.0, +1.0 }, { +1.0, +1.0, +1.0 }, { +1.0, +1.0, -1.0 }, { -1.0, +1.0, -1.0 } }};
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glTranslatef(0.0, 0.0, -10.0);
    glRotatef(rotationX, 1.0, 0.0, 0.0);
    glRotatef(rotationY, 0.0, 1.0, 0.0);
    glRotatef(rotationZ, 0.0, 0.0, 1.0);
    for (int i = 0; i < 6; ++i) {
        glLoadName(i);
        glBegin(GL_QUADS);
        qglColor(faceColors[i]);
        for (int j = 0; j < 4; ++j) {
            glVertex3f(coords[i][j][0], coords[i][j][1], coords[i][j][2]);
        }
```

```
        glEnd();
    }
}
```

draw()에서는 *x, y, z*회전 그리고 faceColors배렬에 보관된 색들을 고려하여 직6면체를 그린다. qglColor()호출을 제외한 모든것이 표준OpenGL이다. 방식에 따라서 OpenGL함수들인 glColor3d() 혹은 glIndex()중 하나를 사용하였다.

```
void Cube::mousePressEvent(QMouseEvent *event)
{
    lastPos = event->pos();
}
void Cube::mouseMoveEvent(QMouseEvent *event)
{
    GLfloat dx = (GLfloat) (event->x() -lastPos.x()) / width();
    GLfloat dy = (GLfloat) (event->y() -lastPos.y()) / height();
    if (event->state() & LeftButton) {
        rotationX += 180 * dy;
        rotationY += 180 * dx;
        updateGL();
    } else if (event->state() & RightButton) {
        rotationX += 180 * dy;
        rotationZ += 180 * dx;
        updateGL();
    }
    lastPos = event->pos();
}
```

mousePressEvent()와 mouseMoveEvent()함수들을 QWidget로부터 재정의하여 사용자가 보기를 찰칵하고 끌기하여 회전할수 있게 한다. 왼쪽 마우스단추는 사용자가 *x*와 *y*축주위로 회전할수 있게 하고 오른쪽 마우스단추는 x와 z축주위를 회전할수 있게 한다.

rotationX, rotationY 혹은 rotationZ변수들을 수정한 다음 updateGL()을 호출하여 배경을 다시 그린다.

```
void Cube::mouseDoubleClickEvent(QMouseEvent *event)
{
    int face = faceAtPosition(event->pos());if (face != -1) {
        QColor color = QColorDialog::getColor(faceColors[face], this);
        if (color.isValid()) {
```

```
                faceColors[face] = color;
                updateGL();
            }
        }
    }
```

mouseDoubleClickEvent()을 QWidget로부터 재정의하여 사용자가 립방체의 한 면을 두번 찰칵하여 면의 색을 설정하게 한다. 비공개함수 faceAtPosition()를 호출하여 유표아래에 배치된 립방체면을 결정한다. 한 면을 두번 찰칵하면 QColorDialog::getColor()를 호출하여 그 면의 새로운 색을 얻는다. 그다음 새로운 색으로 faceColors배렬을 갱신하고 updateGL()를 호출하여 배경을 다시 그린다.

```
int Cube::faceAtPosition(const QPoint &pos)
{
    const int MaxSize = 512;
    GLuint buffer[MaxSize];
    GLint viewport[4];
    glGetIntegerv(GL_VIEWPORT, viewport);
    glSelectBuffer(MaxSize, buffer);
    glRenderMode(GL_SELECT);
    glInitNames();
    glPushName(0);
    glMatrixMode(GL_PROJECTION);
    glPushMatrix();
    glLoadIdentity();
    gluPickMatrix((GLdouble)pos.x(), (GLdouble) (viewport[3] -pos.y()), 5.0, 5.0, viewport);
    GLfloat x = (GLfloat)width() / height();
    glFrustum(-x, x, -1.0, 1.0, 4.0, 15.0);
    draw();
    glMatrixMode(GL_PROJECTION);
    glPopMatrix();
    if (!glRenderMode(GL_RENDER))
        return -1;
    return buffer[3];
}
```

faceAtPosition()함수는 창문부품우에서 일정한 위치에 있는 면의 번호를 돌려주며 그 위치에 면이 없으면 –1을 돌려준다. OpenGL에서 이것을 결정하는 코드는 좀 복잡하다. 본질적으

로 OpenGL의 포착능력의 우점을 리용하도록 GL_SELECT방식으로 배경을 그리고 OpenGL히트(hit)기록으로부터 면번호(이름)를 얻는다.

여기에 main.cpp이 있다.

```
#include <qapplication.h>
#include "cube.h"
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    if (!QGLFormat::hasOpenGL())
        qFatal("This system has no OpenGL support");
    Cube cube;
    cube.setCaption(QObject::tr("Cube"));
    cube.resize(300, 300);
    app.setMainWidget(&cube);
    cube.show();
    return app.exec();
}
```

사용자의 체계가 OpenGL을 유지하지 않으면 콘솔에 오유통보문을 출력하고 Qt의 qFatal()대역함수에 의하여 중지한다.

응용프로그람을 QGL 및 OpenGL서고들에 련결할 때 .pro파일이 이 항목을 요구한다.

```
CONFIG += opengl
```

이로서 Cube응용프로그람을 끝낸다.(QGL모듈에 대한 더 자세한 정보는 QGLWidget, QGLFormat, QGLContext, QGLColormap의 방조문서를 보시오.)

제9장. 끌어다놓기

끌어다놓기(drag and drop)는 하나의 응용프로그람안에서 혹은 서로 다른 응용프로그람들사이에 정보를 전송하는 직관적인 방법이다. 이것은 흔히 자료의 이동 및 복사를 위한 오려둡판기능과 함께 제공된다.

이 장에서는 Qt응용프로그람에 끌어다놓기 기능을 추가하는 방법을 보여주는것으로 시작한다. 그다음 끌어다놓기코드를 재리용하여 오려둡판기능을 실현한다. 이 코드재리용은 두 기구가 각이한 형식으로 자료를 제공하는 추상기초클라스인 QMimeSource에 기초하므로 가능하다.

제1절. 끌어다놓기의 허용

끌어다놓기는 2개의 다른 작용 끌기(dragging)와 놓기(dropping)로 이루어진다. 창문부품들은 끌기자리로서, 놓기자리로서 혹은 끌기와 놓기자리로서 선택될수 있다.

끌어다놓기는 응용프로그람들사이에서 자료를 전송하기 위한 좋은 방법이다. 그러나 일부 경우에는 Qt의 끌어다놓기기능을 사용하지 않고 끌어다놓기를 실현할수 있다. 수행하려는 일이 한개 응용프로그람의 하나의 창문부품안에서 이동하는것이라면 창문부품의 마우스사건처리함수들을 재정의하는것이 훨씬 더 간단하다. 이것은 8장의 DiagramView창문부품에서 사용한 수법이다.

첫 실례는 Qt응용프로그람이 다른 응용프로그람에 의해 초기화된 끌기를 받아들이게 하는 방법을 보여준다. Qt응용프로그람은 QTextEdit를 중심창문부품으로 가지는 기본창문이다. 사용자가 탁상이나 파일열람기로부터 파일을 끌고가서 응용프로그람에서 놓을 때 응용프로그람은 QTextEdit에로 파일을 적재한다.

여기에 MainWindow클라스의 정의가 있다.

```
class MainWindow : public QMainWindow
{
    Q_OBJECT
public:
    MainWindow(QWidget *parent = 0, const char *name = 0);
protected:
    void dragEnterEvent(QDragEnterEvent *event);
    void dropEvent(QDropEvent *event);
private:
    bool readFile(const QString &fileName);
    QString strippedName(const QString &fullFileName);
    QTextEdit *textEdit;
```

```
};
```

MainWindow클라스는 QWidget로부터 dragEnterEvent()와 dropEvent()를 재정의한다. 실례의 목적이 끌어다놓기를 보여주는것이므로 기본창문클라스에서 기대하는 기능의 세부를 생략하였다.

```
MainWindow::MainWindow(QWidget *parent, const char *name) : QMainWindow(parent, name)
{
    setCaption(tr("Drag  File"));
    textEdit = new QTextEdit(this);
    setCentralWidget(textEdit);
    textEdit->viewport() ->setAcceptDrops (false);
    setAcceptDrops(true);
}
```

구성자에서는 QTextEdit를 창조하여 중심창문부품으로 설정한다. QTextEdit의 보기구역에서 놓기를 금지하고 기본창문에서 놓기를 허용한다.

QTextEdit에서 놓기를 금지하는 리유는 MainWindow파생클라스에서 자체로 끌어다놓기를 처리하려는데 있다. 기정으로 QTextEdit는 다른 응용프로그람들로부터 본문끌기를 받아들이고 사용자가 그우에 파일을 놓으면 본문에 파일이름을 삽입한다. 파일이름이 아니라 파일의 전체내용을 놓으려고 하므로 QTextEdit의 끌어다놓기기능을 사용할수 없고 자체로 실현해야 한다.

놓기사건들은 자식으로부터 부모로 전달되므로 MainWindow안에서 QTextEdit용의 놓기사건들을 비롯하여 전체 기본창문용 놓기사건들을 얻는다.

```
void MainWindow::dragEnterEvent(QDragEnterEvent  *event)
{
    event->accept(QUriDrag::canDecode(event));
}
```

dragEnterEvent()는 사용자가 객체를 창문부품우로 끌고갈 때 호출된다. 사건에 대하여 accept(true)를 호출하면 사용자가 놓기객체를 이 창문부품우에 놓을수 있다는것을 가리킨다. accept(false)를 호출하면 창문부품이 끌기를 받아들일수 없다는것을 가리킨다. Qt는 자동적으로 유표를 변경하여 그 창문부품이 타당한 놓기자리인가 아닌가를 사용자에게 알려준다.

여기서는 사용자가 파일들을 끌수 있게만 하려고 한다. 그러기 위하여 파일끌기를 처리하는 Qt클라스인 QUriDrag에게 끌고가는 객체를 해석할수 있는가 묻는다. 이 클라스는 HTTP와 FTP경로와 같은 만능자원식별자(universal resource identifier, URI)에 더 일반적으로 사용되므로 QUriDrag라고 부른다.

```
void MainWindow::dropEvent(QDropEvent  *event)
{
```

```
        QStringList fileNames;
        if (QUriDrag::decodeLocalFiles(event, fileNames)) {
            if (readFile(fileNames[0]))
                setCaption(tr("%1 -Drag File") .arg(strippedName(fileNames[0])));
        }
    }
```

dropEvent()는 사용자가 객체를 창문부품우에 놓을 때 호출된다. 정적함수 QUriDrag::decodeLocalFiles()를 호출하여 사용자가 끌고있는 파일이름목록을 얻고 목록안의 첫 파일을 읽어들인다. (둘째 인수는 비상수참고로 넘어간다.) 일반적으로 사용자들은 한번에 하나의 파일만 끌기하지만 하나의 선택을 끌기하여 여러개의 파일을 끌기할수 있다.

또한 QWidget는 dragMoveEvent()와 dragLeaveEvent()를 제공하지만 대부분의 응용프로그람들에서 이것들을 재정의할 필요는 없다.

둘째 실례는 끌기를 초기화하고 놓기를 받아들이는 방법을 설명한다. 끌어다놓기를 유지하는 QListBox의 파생클라스를 창조하고 그림 9-1에 보여주는 Project Chooser응용프로그람의 부분품으로 사용한다.

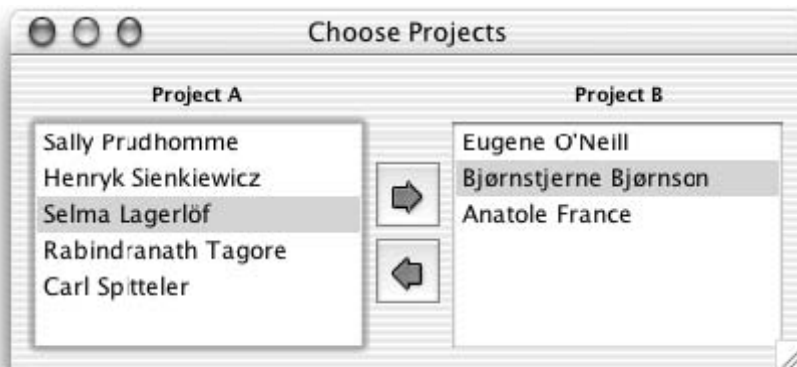



그림 9-1. Project Chooser응용프로그람

Project Chooser응용프로그람은 사용자에게 이름들이 들어있는 2개의 목록칸을 표시한다. 매개의 목록칸은 하나의 프로젝트를 표시한다. 사용자는 목록칸안에 이름들을 끌어다넣음으로써 사람을 한 프로젝트에서 다른 프로젝트로 옮길수 있다.

끌어다놓기코드는 모두 QListBox의 파생클라스에 배치된다. 여기에 클라스정의가 있다.

```
class ProjectView : public QListBox
{
    Q_OBJECT
public:
    ProjectView(QWidget *parent, const char *name = 0);
protected:
```

```
        void contentsMousePressEvent(QMouseEvent *event);
        void contentsMouseMoveEvent(QMouseEvent *event);
        void contentsDragEnterEvent(QDragEnterEvent *event);
        void contentsDropEvent(QDropEvent *event);
    private:
        void startDrag();
        QPoint dragPos;
    };
```

ProjectView는 QListBox의 기초클라스인 QScrollView에서 선언된 4개의 사건처리함수들을 재정의한다.

```
    ProjectView::ProjectView(QWidget *parent, const char *name) : QListBox(parent, name)
    {
        viewport()->setAcceptDrops(true);
    }
```

구성자에서는 QScrollView보기구역우에서의 놓기를 허용한다.

```
    void ProjectView::contentsMousePressEvent(QMouseEvent *event)
    {
       if (event->button() == LeftButton)
            dragPos = event->pos();
       QListBox::contentsMousePressEvent(event);
    }
```

사용자가 왼쪽 마우스단추를 찰칵할 때 마우스위치를 dragPos비공개변수에 보관한다. contentsMousePressEvent()의 QListBox의 실현을 호출하여 QListBox가 보통과 같이 마우스누르기사건들을 처리할 기회를 가지도록 한다.

```
    void ProjectView::contentsMouseMoveEvent(QMouseEvent *event)
    {
       if (event->state() & LeftButton) {
           int distance = (event->pos() -dragPos).manhattanLength();
           if (distance > QApplication::startDragDistance())
               startDrag();
       }
       QListBox::contentsMouseMoveEvent(event);
    }
```

사용자가 왼쪽 마우스단추를 누른 상태에서 마우스유표를 이동할 때 끌기를 시작한다. 이전의 마우스위치와 현재 왼쪽 마우스단추를 누른 위치사이의 거리를 계산한다.

그 거리가 QApplication의 주어진 끌기시작거리(보통 4화소)보다 크면 비공개함수 startDrag()를 호출하여 끌기를 시작한다. 이것은 사용자의 손이 떨리는것으로 인한 끌기의 시작을 피한다.

```
void ProjectView::startDrag()
{
    QString person = currentText();
    if (!person.isEmpty()) {
        QTextDrag *drag = new QTextDrag(person, this);
        drag->setSubtype("x-person");
        drag->setPixmap(QPixmap::fromMimeSource("person.png"));
        drag->drag();
    }
}
```

startDrag()에서는 부모로서 this를 가지는 QTextDrag형의 객체를 창조한다. QTextDrag클라스는 본문을 전송하기 위한 끌어다놓기객체를 표시한다. 이것은 Qt가 제공하는 끌기객체들의 사전정의형들중의 하나이고 다른것들로서 QImageDrag, QColorDrag, QUriDrag를 포함한다. 또한 픽스매프를 설정하여 끌기를 표시한다. 픽스매프는 끌기가 진행되는동안 유표를 따르는 작은 그림기호이다.

setSubtype()를 호출하여 객체의 MIME형의 보조형을 x-person으로 설정한다. 이것은 객체의 완전한 MIME형을 text/x-person으로 되게 한다. setSubtype()를 호출하지 않았다면 MIME형은 text/plain일수 있다.

표준MIME형들은 IANA (Internet Assigned Numbers Authority)에 의하여 정의된다. 이것들은 형과 사선으로 구분된 보조형으로 이루어진다. text/x-person과 같은 비표준형을 창조할 때 보조형에 x-앞붙이를 추가할것을 권고한다. MIME형들은 오려둠판과 끌어다놓기체계에서 각이한 형의 자료들을 식별하는데 쓰인다.

drag()호출은 끌기조작을 시작한다. 호출후에 QTextDrag객체는 끌기조작이 끝날 때까지 존재한다. Qt는 끌기객체의 소유자를 가지며 그것이 더는 필요하지 않은데 놓지 않은 경우에도 삭제한다.

```
void ProjectView::contentsDragEnterEvent(QDragEnterEvent *event)
{
    event->accept(event->provides("text/x-person"));
}
```

ProjectView창문부품은 text/x-person형의 끌기를 시작할뿐아니라 그러한 끌기를 받아들인다. 끌기가 창문부품에 들어갈 때 정확한 MIME형을 가지는가 검사하고 없으면 그것을 거부한다.

```
void ProjectView::contentsDropEvent(QDropEvent *event)
```

```
{
      QString person;
      if (QTextDrag::decode(event, person)) {
            QWidget *fromWidget = event->source();
            if (fromWidget && fromWidget != this && fromWidget->inherits("ProjectView")) {
                  ProjectView *fromProject = (ProjectView *) fromWidget;
                  QListBoxItem *item = fromProject->findItem(person, ExactMatch);
                  delete item;
                  insertItem(person);
            }
      }
}
```

contentsDropEvent()에서는 QTextDrag::decode()함수를 사용하여 끌기에 의해 날라온 본문을 꺼낸다. QDropEvent::source()함수는 끌기를 시작하는 창문부품이 같은 응용프로그람의 부분이라면 그것의 지적자를 돌려준다. 원천창문부품이 목표창문부품과 다른데 ProjectView이면 delete를 호출하여 원천창문부품으로부터 항목을 삭제하고 새로운 항목을 목표창문부품에 삽입한다.

제2절. 사용자정의끌기형의 유지

지금까지의 실례들에서는 미리 정의된 Qt클라스들에 기초하여 끌기자료를 보관해왔다. 례를 들면 파일끌기에 QUriDrag, 본문끌기에 QTextDrag를 사용하였다. 이 두개의 클라스들은 모든 끌기객체들의 기초클라스인 QDragObject를 계승한다. QDragObject자체는 MIME형 자료를 제공하는 추상클라스 QMimeSource를 계승한다.

본문, 화상, URI, 혹은 색을 끌기하려면 Qt의 QTextDrag, QImageDrag, QUriDrag, QColorDrag 클라스들을 사용할수 있다. 그러나 사용자정의자료를 끌기하려면 미리 정의된 어느 클라스도 적합지 않으므로 다음의 두가지 방법들중 하나를 선택해야 한다.

· 끌기를 QStoredDrag객체에 2진자료로서 보관할수 있다.

· QDragObject의 파생클라스를 만들고 한조의 가상함수들을 재정의하여 자체의 끌기클라스를 창조할수 있다.

QStoredDrag는 임의의 2진자료를 보관하게 하므로 어떤 MIME형에서나 사용될수 있다. 례를 들면 ASDF형식으로 자료를 보관하는 2진파일의 내용을 가지고 끌기를 시작하려면 다음의 코드를 사용할수 있다.

```
void MyWidget::startDrag()
{
      QByteArray data = toAsdf();
      if (!data.isEmpty()) {
```

```
        QStoredDrag *drag = new QStoredDrag("octet-stream/x-asdf", this);
        drag->setEncodedData(data);
        drag->setPixmap(QPixmap::fromMimeSource("asdf.png"));
        drag->drag();
    }
}
```

QStoredDrag에서 한가지 결함은 오직 하나의 MIME형을 보관할수 있는것이다. 같은 응용프로그람안에서 혹은 같은 응용프로그람의 여러개의 실례들사이에서 끌어다놓기를 수행한다면 이것은 그리 문제로 되지 않는다. 그러나 다른 응용프로그람들과 간단하게 교제하려고 한다면 하나의 MIME형으로도 충분하다.

또 하나의 결함은 마감에 끌기를 받아들이지 않는 경우에도 자료구조를 QByteArray로 변환할 필요가 있는것이다. 자료가 크면 이것은 응용프로그람이 매우 느리게 할수 있다. 사용자가 실제로 끌기객체를 놓을 때에만 자료변환을 수행하는것이 더 좋다.

이 두 문제에 대한 해결대책은 QDragObject의 파생클라스를 만들고 Qt에서 끌기정보를 얻는데 쓰이는 두개의 가상함수들인 format()와 encodedData()를 재정의하는것이다. 그 동작방법을 보여주기 위하여 직4각형의 QTable선택에 하나이상의 세포의 내용을 보관하는 CellDrag 클라스를 개발한다.

```
class CellDrag : public QDragObject
{
public:
    CellDrag(const QString &text, QWidget *parent = 0,
    const char *name = 0);
    const char *format(int index) const;
    QByteArray encodedData(const char *format) const;
    static bool canDecode(const QMimeSource *source);
    static bool decode(const QMimeSource *source, QString &str);
private:
    QString toCsv() const;QString toHtml() const;
    QString plainText;
};
```

CellDrag클라스는 QDragObject를 계승한다. 끌기에서 실제로 문제가 있는 2개의 함수는 format()와 encodedData()이다. 엄격히 필요하지 않지만 canDecode()와 decode()정적함수들을 제공하여 놓기할 때 자료를 꺼내는것이 편리하다.

```
CellDrag::CellDrag(const QString &text, QWidget *parent, const char *name)
    : QDragObject(parent, name)
```

```
{
    plainText = text;
}
```

CellDrag구성자는 끌고있는 세포들의 내용을 표시하는 문자렬을 받아들인다. 문자렬은 4장에서 표계산프로그람에 오려둠판기능을 추가할 때 사용한 《타브와 새행》평본문서식으로 되여있다.

```
const char *CellDrag::format(int index) const
{
    switch (index) {
        case 0:
            return "text/csv";
        case 1:
            return "text/html";
        case 2:
            return "text/plain";
        default:
            return 0;
    }
}
```

format()함수는 끌기에 유지된 각이한 MIME형을 돌려주도록 QMimeSource로부터 재정의된다. 3가지 형 즉 반점구분값(comma-separated values, CSV), HTML, 평본문을 유지한다.

Qt가 끌기에 의해 어떤 MIME형들이 제공되는가를 결정할 때 format()가 null지적자를 돌려줄 때까지 첨수파라메터 0, 1, 2, ...을 넘기여 format()를 호출한다.

서식의 정확한 순서는 보통 아무런 관계도 없으나 가장 좋은 서식을 우선 넣는것이 좋다. 많은 서식을 유지하는 응용프로그람들은 흔히 처음으로 일치하는 서식을 리용한다.

```
QByteArray CellDrag::encodedData(const char *format) const
{
    QByteArray data;
    QTextOStream out(data);
    if (qstrcmp(format, "text/CSV") == 0) {
        out << toCsv();
    } else if (qstrcmp(format, "text/html") == 0) {
        out << toHTML();
    } else if (qstrcmp(format, "text/plain") == 0) {
        out << plainText;
```

```
        }
        return data;
    }
```

encodedData()함수는 주어진 MIME형을 위한 자료를 돌려준다. 서식파라메터의 값은 보통 format()가 돌려준 하나의 문자렬이지만 모든 응용프로그람들이 미리 format()에 대하여 MIME형을 검사하지 않으므로 그것을 가정할수 없다. Qt응용프로그람들에서 이 검사는 보통 처음에 수행한것처럼 QDragEnterEvent나 QDragMoveEvent에 대하여 provides()를 호출하여 수행된다.

QString을 QByteArray로 변환하는 가장 좋은 수법은 QTextStream을 사용하는것이다. 문자렬이 비ASCII문자들을 포함한다면 QTextStream은 부호화가 국부8비트부호화라고 가정한다. (대부분의 유럽나라들에서 이것은 ISO 8859-1 혹은 ISO 8859-15를 의미한다. 자세한 내용은 15장 참고.) 15장에서 설명하는것처럼 흐름에 대하여 setEncoding() 혹은 setCodec()를 호출하여 다른 부호화를 사용하도록 할수 있다.

```
    QString CellDrag::toCsv() const
    {
        QString out = plainText;
        out.replace("\\", "\\\\");
        out.replace("\"", "\\\"");
        out.replace("\t", "\", \"");
        out.replace("\n", "\"\n\"");
        out.prepend("\"");
        out.append("\"");
        return out;
    }
    QString CellDrag::toHtml() const
    {
        QString out = QStyleSheet::escape(plainText);
        out.replace("\t", "<td>");
        out.replace("\n", "\n<tr><td>");
        out.prepend("<table>\n<tr><td>");
        out.append("\n</table>");
        return out;
    }
```

toCsv()와 toHtml()함수들은 《타브와 새행》문자렬을 CSV 혹은 HTML문자렬로 변환한다. 례를 들면 자료

Red Green Blue

Cyan Yellow Magenta

는

"Red", "Green", "Blue"

"Cyan", "Yellow", "Magenta"

로 변환되거나

<table> <tr><td>Red<td>Green<td>Blue <tr><td>Cyan<td>Yellow<td>Magenta </table>

로 변환된다.

변환은 가장 간단한 방법 즉 QString::replace()에 의해 수행된다. HTML특수문자들을 확장하기 위하여 QStyleSheet::escape()정적편의함수를 사용한다.

```
bool CellDrag::canDecode(const QMimeSource *source)
{
    return source->provides("text/plain");
}
```

canDecode()함수는 주어진 끌기를 해석할수 있으면 true, 그렇지 않으면 false를 돌려준다. 최대의 유연성을 위하여 그 인수는 하나의 QMimeSource이다. QMimeSource클라스는 QDragObject, QDragEnterEvent, QDragMoveEvent, QDropEvent의 기초클라스이다.

자료를 3가지 서로 다른 서식으로 제공하지만 오직 평본문만 받아들인다. 보통 그 리유는 평문이면 충분하기때문이다. 사용자가 세포들을 QTable로부터 HTML편집기에로 끌기하면 세포들을 HTML표로 변환하려고 한다. 그러나 사용자가 임의의 HTML을 QTable로 끌기하면 그것을 받아들이려고 하지 않는다.

```
bool CellDrag::decode(const QMimeSource *source, QString &str)
{
    QByteArray data = source->encodedData("text/plain");
    str = QString::fromLocal8Bit((const char *)data, data.size());
    return !str.isEmpty();
}
```

끝으로 decode()함수는 text/plain자료를 QString으로 변환한다 다시 본문이 국부8비트부호화로 부호화된다고 가정한다.

정확한 부호화를 사용하려는것이 확실하면 text/plain MIME형의 charset파라메터를 리용하여 명시적인 부호화를 지정할수 있다. 여기에 몇가지 실례가 있다.

```
text/plain;charset=US-ASCII
text/plain;charset=ISO-8859-1
text/plain;charset=Shift_JIS
```

QTextDrag를 사용할 때 항상 UTF-8, UCS-2 (UTF-16), US-ASCII, 그리고 국부8비트부호화를 반출하고 다른 부호화로부터 놓기를 받아들인다. 이것을 고려하여 QTextDrag::decode()를 호출

하는것으로 CellDrag::decode()를 간단히 실현하는것이 더 깨끗하다. 그러나 이 수법을 리용해도 후에 확장하여 평본문외에 다른 형의 끌기(실례로 CSV끌기)를 해석하려고 하는 경우에는 여전히 CellDrag::decode()를 QTextDrag::decode()와 구별하여 제공한다.

이제는 CellDrag클라스가 완성되였다. 그것을 사용하기 위하여 QTable과 통합한다. QTable이 이미 거의 대부분의 작업을 수행한다. 우리가 해야 할 일은 그 파생클라스를 만들고 파생클라스의 구성자에서 setDragEnabled(true)를 호출하고 CellDrag를 돌려주도록 QTable::dragObject()를 재정의하는것이다. 여기에 실례가 있다.

```
QDragObject *MyTable::dragObject()
{
    return new CellDrag(selectionAsString(), this);
}
```

selectionAsString()의 코드는 Spreadsheet::copy()함수의 핵심과 같으므로 여기서 보여주지 않는다.

QTable에 놓기기능을 추가하려면 Project Chooser응용프로그람에서와 같은 방법으로 contentsDragEnterEvent()와 contentsDropEvent()를 재정의하여야 한다.

제3절. 오려둠판의 고급한 조종

대부분의 응용프로그람들은 Qt의 내부오려둠판조종기능을 여러가지 방법으로 리용한다. 례를 들면 QTextEdit클라스는 cut(), copy(), paste()처리부들에 의하여 Ctrl+X, Ctrl+C, Ctrl+V기능들을 제공하므로 코드를 전혀 추가하지 않아도 된다.

자체의 클라스들을 쓸 때 응용프로그람의 QClipboard객체의 지적자를 돌려주는 QApplication::clipboard()를 통하여 오려둠판을 호출할수 있다. 체계오려둠판의 조종은 간단하다. setText(), setImage() 혹은 setPixmap()를 호출하여 자료를 오려둠판에 넣고 text(), image() 혹은 pixmap()를 호출하여 자료를 얻는다. 이미 4장의 표계산프로그람과 8장의 Diagram프로그람에서 오려둠판의 실례들을 보았다.

어떤 응용프로그람들에서는 내부기능만으로 충분하지 못하다. 례를 들면 본문이나 화상이 아닌 자료를 주려고 한다. 또는 다른 응용프로그람들과의 최대한의 호상운영을 위하여 서로 다른 수많은 서식의 자료를 제공하려고 할수 있다. 문제는 초기에 끌어다놓기에서 제기된것과 비슷하며 그 대답도 비슷하다. 즉 QMimeSource의 파생클라스를 만들고 format()와 encodedData()를 재정의한다.

응용프로그람이 끌어다놓기기능을 가지고있다면 QDragObject의 사용자정의파생클라스를 간단히 재리용하고 setData()함수에 의하여 그것을 오려둠판에 넣을수 있다. QDragObject가 QMimeSource를 계승하고 오려둠판이 QMimeSource들을 리해하면 이것은 원만히 작업한다.

례를 들면 여기에 QTable의 파생클라스의 copy()함수를 실현하는 방법이 있다.

```
void MyTable::copy()
{
```

```
        QApplication::clipboard()->setData(dragObject());
}
```

앞절의 마감에 선택된 세포들의 내용을 보관하는 CellDrag를 돌려주도록 dragObject()를 실현하였다.

자료를 얻기 위하여서는 오려둠판에 대하여 data()를 호출한다. 여기에 QTable파생클라스의 paste()함수를 실현하는 방법이 있다.

```
void MyTable::paste()
{
    QMimeSource *source = QApplication::clipboard() ->data();
    if (CellDrag::canDecode(source)) {
        QString str;
        CellDrag::decode(source, str);
        performPaste(str);
    }
}
```

performPaste()는 본질적으로 4장에서 제시한 Spreadsheet::paste()함수와 같다.

이것은 사용자정의QMimeSource중에서 사용자정의형의 오려둠판기능을 추가하는데 필요한 전부이다.

X11오려둠판은 Windows나 Mac OS X에서 사용할수 없는 추가적인 기능을 제공한다. X11에서는 보통 3개단추마우스의 중간단추를 찰칵하여 선택내용을 붙이기할수 있다. 이것은 개별적인 《선택》오려둠판에 의해 수행된다. 자기의 창문부품들이 표준오려둠판은 물론 이러한 종류의 오려둠판을 유지하려고 한다면 여러가지 오려둠판호출에 추가인수로서 QClipboard::Selection을 넘겨야 한다. 례를 들면 여기에 본문편집기에서 중간마우스단추에 의한 붙이기기능을 유지하도록 mouseReleaseEvent()를 재정의하는 방법이 있다.

```
void MyTextEditor::mouseReleaseEvent(QMouseEvent *event)
{
    QClipboard *clipboard = QApplication::clipboard();
    if (event->button() == MidButton && clipboard->supportsSelection()) {
        QString text = clipboard->text(QClipboard::Selection);
        pasteText(text);
    }
}
```

X11에서 supportsSelection()함수는 true를 돌려준다. 다른 가동환경들에서는 false를 돌려준다.

제10장. 입력과 출력

이 장에서는 파일의 읽기와 쓰기, 파일체계의 항행, 외부프로그람들과의 호상교제에 대하여 설명한다.

Qt의 QDataStream과 QTextStream클라스들은 파일을 간단히 읽고 쓰게 한다. 이 클라스들은 바이트순서화와 본문부호화와 같은 문제들을 고려하여 각이한 가동환경에서 실행하는 Qt응용프로그람들이 파일들을 읽고 쓸수 있게 한다.

수많은 응용프로그람들은 등록부를 항행하거나 파일에 대한 정보를 얻어야 한다. Qt의 QDir와 QFileInfo클라스들이 이것을 가능하게 한다.

일부 상황에서는 GUI프로그람내에서 외부프로그람들을 실행해야 한다. Qt의 QProcess클라스는 GUI응답성을 유지하면서 외부프로그람들을 신호들과 비동기적으로 실행되게 한다.

제1절. 2진자료의 읽기와 쓰기

QDataStream에 의한 2진자료의 읽기와 쓰기는 Qt에서 사용자정의자료를 적재하고 보관하는 가장 간단한 방법이다. QDataStream은 QByteArray, QFont, QImage, QMap<K, T>, QPixmap, QString, QValueList<T>, QVariant를 비롯한 수많은 Qt자료형들을 유지한다.

2진자료의 취급방법을 보여주기 위하여 2개의 실례클라스 즉 Drawing과 Gallery를 리용한다. Drawing클라스는 그림에 대한 기초정보(저자의 이름, 제목, 창작년도)를 보유하고 Gallery클라스는 Drawing들의 목록을 보관한다.

Gallery클라스로부터 시작하자.

```
class Gallery : public QObject {
public:
    bool loadBinary(const QString &fileName);
    bool saveBinary(const QString &fileName);
    ...
private:
    enum { MagicNumber = 0x98c58f26 };
    void writeToStream(QDataStream &out);
    void readFromStream(QDataStream &in);
    void error(const QFile &file, const QString &message);
    void ioError(const QFile &file, const QString &message);
    QByteArray getData();
    void setData(const QByteArray &data);
    QString toString();
    std::list<Drawing> drawings;
```

```
};
```

Gallery클라스는 자료를 보관하고 적재하기 위한 공개함수들을 포함한다. 자료는 drawings 자료성원에 보관된 그림들의 목록이다. 비공개함수들은 그것들을 사용할 때 설명한다.

여기에 Gallery의 그림들을 2진자료로 보관하기 위한 간단한 함수가 있다.

```
bool Gallery::saveBinary(const QString &fileName)
{
    QFile file(fileName);
    if (!file.open(IO_WriteOnly)) {
        ioError(file, tr("Cannot open file %1 for writing"));
        return false;
    }
    QDataStream out(&file);
    out.setVersion(5);
    out << (Q_UINT32)MagicNumber;
    writeToStream(out);
    if (file.status() != IO_Ok) {
        ioError(file, tr("Error writing to file %1"));
        return false;
    }
    return true;
}
```

파일을 열고 그 파일을 QDataStream의 목표로 만든다. QDataStream의 판을 5 (Qt 3.2에서 제일 최근 판)로 설정한다. 판번호는 Qt자료형들을 표시하는 방법에 영향을 준다. C++기본자료형들은 늘 같은 방법으로 표시된다.

그다음 Gallery파일형식을 식별하는 수(MagicNumber)를 출력한다. 모든 가동환경에서 수가 32비트옹근수로 씌여진다는것을 담보하기 위하여 수를 정확히 32비트로 만드는 자료형인 Q_UINT32로 강제변환한다.

파일본체는 writeToStream()비공개함수에 의해 써넣어진다. 파일을 정확히 닫을 필요가 없고 이것은 QFile변수가 함수끝에서 리용범위밖으로 벗어날 때 자동적으로 수행된다.

writeToStream()호출후에 QFile장치의 상태를 검사한다. 오유가 있으면 ioError()를 호출하여 사용자에게 통보창을 표시하고 false를 돌려준다.

```
void Gallery::ioError(const QFile &file, const QString &message)
{
    error(file, message + ": " + file.errorString());
}
```

ioError()함수는 일반적인 error()함수를 호출한다.

```
void Gallery::error(const QFile &file, const QString &message)
{
     QMessageBox::warning(0, tr("Gallery"), message.arg(file.name()));
}
```

이제는 writeToStream()함수를 고찰하자.

```
void Gallery::writeToStream(QDataStream &out)
{
    list<Drawing>::const_iterator it = drawings.begin();
    while (it != drawings.end()) {
         out << *it;++it;
    }
}
```

writeToStream()함수는 Gallery의 그림들을 모두 순환하면서 Drawing클라스의 << 연산자에 기초하여 주어진 흐름에 그것들을 출력한다. 그림들을 보관하는데 list<Drawing>대신에 QValueList<Drawing>를 리용한다면 순환을 생략하고 간단히 다음과 같이 쓰면 된다.

```
out << drawings;
```

QValueList<T>가 흐름에 흐를 때 목록에 보관된 매개 항목은 항목형의 <<연산자에 의하여 출력된다.

```
QDataStream &operator<<(QDataStream &out, const Drawing &drawing)
{
    out << drawing.myTitle << drawing.myArtist << drawing.myYear;
    return out;
}
```

Drawing을 출력하기 위해서는 세개 비공개성원변수들 즉 myTitle, myArtist, myYear를 간단히 출력한다. 그러기 위하여 operator<<()를 Drawing의 동료로서 선언할 필요가 있다. 함수의 끝에서 흐름을 돌려준다. 이것은 C++의 일반적인 관례로서 출력흐름과 함께 여러개의 << 연산자들을 련이어 쓸수 있게 한다. 례를 들면

```
out << drawing1 << drawing2 << drawing3;
```

Drawing클라스의 정의는 다음과 같다.

```
class Drawing
{
    friend QDataStream &operator<<(QDataStream &, const Drawing &);
    friend QDataStream &operator>>(QDataStream &, Drawing &);
public:
```

```
    Drawing() { myYear = 0;}
    Drawing(const QString &title, const QString &artist, int year)
       { myTitle = title;myArtist = artist;myYear = year; }
    QString title() const { return myTitle; }
    void setTitle(const QString &title) { myTitle = title; }
    QString artist() const { return myArtist; }
    void setArtist(const QString &artist) { myArtist = artist; }
    int year() const { return myYear;}
    void setYear(int year) { myYear = year; }
private:
    QString myTitle;QString myArtist;int myYear;
};
```

그러면 Gallery파일로부터 자료를 읽어들이는 방법을 고찰하자.

```
bool Gallery::loadBinary(const QString &fileName)
{
    QFile file(fileName);
    if (!file.open(IO_ReadOnly)) {
        ioError(file, tr("Cannot open file %1 for reading"));
        return false;
    }
    QDataStream in (&file);
    in.setVersion(5);
    Q_UINT32 magic;
    in >> magic;
    if (magic != MagicNumber) {
        error(file, tr("File %1 is not a Gallery file"));
        return false;
    }
    readFromStream(in);
    if (file.status() != IO_Ok) {
        ioError(file, tr("Error reading from file %1"));
        return false;
    }
    return true;
}
```

읽으려는 파일을 열고 파일로부터 자료를 읽어들이기 위한 QDataStream을 창조한다. QDataStream의 판번호를 5로 설정한다. 그것은 이 판을 써넣는데 사용했기때문이다. 고정판번호 5를 사용하여 응용프로그람이 항상 자료를 읽고 쓸수 있다는것을 담보함으로써 Qt 3.2이후의 판에서 콤파일할수 있게 한다.

우리가 써넣은 식별번호를 읽어들여 그것을 MagicNumber와 비교한다. 이것은 실제로 Gallery파일을 읽어들이고 있다는것을 담보한다. 그다음 readFromStream()함수에 의해 자료를 읽어들인다.

```
void Gallery::readFromStream(QDataStream &in)
{
    drawings.clear();while (!in.atEnd()) {
        Drawing drawing;
        in >> drawing;
        drawings.push_back(drawing);
    }
}
```

readFromStream()에서는 우선 현존자료를 삭제한다. 그다음 >>연산자에 기초하여 한번에 하나의 그림을 읽어들이고 Gallery의 그림목록에 매개를 추가한다. 자료를 보관하는데 list<Drawing>이 아니라 QValueList<Drawing>를 사용한다면 순환없이 모든 그림을 읽어들일수 있다. 즉

```
in >> drawings;
```

QValueList<T>는 항목형의 >>연산자에 기초하여 항목들을 읽어들인다.

```
QDataStream &operator>>(QDataStream &in, Drawing &drawing)
{
    in >> drawing.myTitle >> drawing.myArtist >> drawing.myYear;
    return in;
}
```

>>연산자의 실현은 << 연산자의 실현과 비슷하다. QDataStream를 사용할 때 어떤 종류의 문법해석도 필요없다.

2진자료를 읽고 쓰려고 한다면 QDataStream을 통하여 바이트블로크를 읽고쓰는데 readRawBytes()와 writeRawBytes()를 사용할수 있다. 바이트들앞에 블로크크기가 놓이지 않는다.

기본형(Q_UINT16 혹은 float 등)에 대하여 >>와 <<연산자를 사용하거나 readRawBytes()와 writeRawBytes()를 리용하여 DBF파일과 TEX DVI파일들과 같은 표준2진형식들을 읽어들이고 써넣을수 있다. QDataStream에서 사용되는 기정바이트순서화는 비그엔디안(big-endian)이다. 리틀엔디안(little-endian)으로 자료를 읽고 쓰려면 다음과 같이 호출해야 한다.

```
stream.setByteOrder(QDataStream::LittleEndian);
```

QDataStream이 기본C++자료형을 읽고쓰는데만 사용되고있다면 setVersion()을 사용할 필요가 없다.

파일을 한번에 읽거나 써넣으려면 QDataStream의 리용을 피하고 그대신에 QFile의 writeBlock()와 readAll()함수들을 사용한다. 례를 들면

```
file.writeBlock(getData());
```

이러한 방법으로 써넣은 자료는 바이트렬이다. 자료를 써넣을 때 그것을 구조화하고 자료를 다시 읽어들일 때 그것을 해석한다. Gallery의 비공개getData()함수에 기초하여 QByteArray를 창조하고 거기에 자료를 옮긴다. 자료를 읽어들이는것은 간단하다. 즉

```
setData(file.readAll());
```

QByteArray에 정보를 설정하는데 Gallery의 setData()함수를 사용한다.

QByteArray에 모든 자료를 보관하면 더 많은 기억기를 요구하지만 몇가지 우점이 있다. 례를 들면 Qt의 qCompress()함수를 리용하여 자료를 압축(zlib사용)할수 있다. 즉

```
file.writeBlock(qCompress(getData()));
```

그다음 qUncompress()를 사용하여 자료를 풀수 있다. 즉

```
setData(qUncompress(file.readAll()));
```

getData()와 setData()를 실현하는 한가지 방법은 QByteArray에 대하여 QDataStream을 사용하는것이다. 여기에 getData()가 있다.

```
QByteArray Gallery::getData()
{
    QByteArray data;
    QDataStream out(data, IO_WriteOnly);
    writeToStream(out);
    return data;
}
```

QFile이 아니라 QByteArray에 써넣는 QDataStream을 창조하고 2진자료를 배렬에 채우기 위하여 처음에 사용한 writeToStream()함수를 사용한다.

마찬가지로 setData()함수는 처음에 작성한 readFromStream()함수를 사용한다.

```
void Gallery::setData(const QByteArray &data)
{
    QDataStream in(data, IO_ReadOnly);
    readFromStream(in);
}
```

처음의 실례들에서 흐름의 판을 5로 고정하여 자료를 적재하고 보관하였다. 이 수법은 간단하고 안전하지만 하나의 자그마한 결함이 있다. 즉 새롭거나 갱신된 형식들의 우점을 사용

할수 없다. 례를 들면 Qt의 후판에 QFont부분품(즉 서체의 점크기, 계렬 등)이 새로 추가되였다면 그 부분품은 보관되거나 적재되지 않는다.

하나의 대책은 파일에 QDataStream판번호를 보관하는것이다. 즉

```
QDataStream out(&file);
out << (Q_UINT32)MagicNumber;
out << (Q_UINT16)out.version();
writeToStream(out);
```

이것은 어떤 일이 생겨도 QDataStream의 가장 최근판을 사용하여 자료를 늘 써넣는다는것을 담보한다.

파일을 읽으려고 할 때 식별번호와 흐름판을 읽는다. 즉

```
QDataStream in(&file);
Q_UINT32 magic;
Q_UINT16 streamVersion;
in >> magic >> streamVersion;
if (magic != MagicNumber) {
    error(file, tr("File %1 is not a Gallery file"));
    return false;
} else if ((int)streamVersion > in.version()) {
    error(file, tr("File %1 is from a more recent version of the application"));
    return false;
}
in.setVersion(streamVersion);
readFromStream(in);
```

흐름판이 응용프로그람에 의해 사용된 판보다 작거나 같으면 자료를 읽어들일수 있다. 그렇지 않으면 오유를 알린다.

파일형식이 자체의 판번호를 포함한다면 그것을 흐름판번호대신에 사용할수 있다. 례를 들면 파일형식이 응용프로그람의 1.3판용이라고 가정하자. 그때 자료를 다음과 같이 쓸수 있다.

```
QDataStream out(&file);
out.setVersion(5);
out << (Q_UINT32)MagicNumber;
out << (Q_UINT16)0x0103;
writeToStream(out);
```

자료를 읽어들일 때 응용프로그람의 판번호에 기초하여 어느판의 QDataStream을 사용하는가 결정한다. 즉

```
    QDataStream in(&file);
    Q_UINT32 magic;
    Q_UINT16 appVersion;
    in >> magic >> appVersion;
    if (magic != MagicNumber) {
        error(file, tr("File %1 is not a Gallery file"));
        return false;
    } else if (appVersion > 0x0103) {
        error(file, tr("File %1 is from a more recent version of the application"));
        return false;
    }
    if (appVersion <= 0x0102) {
        in.setVersion(4);
    } else {
        in.setVersion(5);
    }
    readFromStream(in);
```

이 실례에서는 1.2판이전의 응용프로그람들에서 보관한 파일이 자료흐름판 4를 사용한다는것과 1.3판의 응용프로그람에서 보관한 파일들이 자료흐름판 5를 사용한다.

QDataStream판조종방략이 있으면 Qt에 의한 2진자료의 읽고쓰기는 간단하고 믿음성있다.

제2절. 본문의 읽기와 쓰기

Qt는 본문자료를 읽고쓰기 위한 QTextStream클라스를 제공한다. 평본문파일들이나 HTML, XML, 원천파일들과 같은 다른 형식의 파일들을 읽고쓰는데 QTextStream을 사용할수 있다. 이 클라스는 유니코드와 체계의 국부8비트부호화사이의 변환을 고려하며 서로 다른 조작체계들에서 사용되는 서로 다른 행끝처리를 알기 쉽게 진행한다.

QTextStream은 자료의 기본단위로서 QChar를 사용한다. 문자와 문자렬외에 QTextStream은 C++의 기본수값형을 유지하며 그것들과 문자렬사이를 변환한다.

QTextStream의 사용법을 보여주기 위하여 앞절의 Gallery실례를 계속한다. 여기에 Gallery로부터 그림자료들을 보관하는 saveText()함수의 코드가 있다.

```
bool Gallery::saveText(const QString &fileName)
{
    QFile file(fileName);
    if (!file.open(IO_WriteOnly | IO_Translate)) {
        ioError(file, tr("Cannot open file %1 for writing"));
        return false;
```

```
    }
    QTextStream out(&file);
    out.setEncoding(QTextStream::UnicodeUTF8);
    list<Drawing>::const_iterator it = drawings.begin();
    while (it != drawings.end()) {
        out << *it;
        ++it;
    }
    if (file.status() != IO_Ok) {
        ioError(file, tr("Error writing to file %1"));
        return false;
    }
    return true;
}
```

IO_Translate기발을 사용하여 파일을 열 때 새행문자를 목표가동환경의 정확한 문자렬(Windows에서 "\r\n", Mac OS X에서 "\r")로 변환한다. 그다음 전체 유니코드문자모임을 표시할 수 있는 ASCII호환부호화인 UTF-8로 부호화를 설정한다. (유니코드에 대한 자세한 정보는 15장 참고.) 출력을 조종하려면 <<연산자에 기초하여 Gallery안의 매개 그림을 순환한다.

```
QTextStream &operator<<(QTextStream &out, const Drawing &drawing)
{
    out << drawing.myTitle << ":" << drawing.myArtist << ":"
        << drawing.myYear << endl;
    return out;
}
```

그림을 써넣을 때에는 하나의 두점을 리용하여 그림의 제목과 저자의 이름을 분리하고 다른 두점으로 저자의 이름과 년도를 분리하며 새행으로 자료를 끝낸다. 제목과 저자의 이름은 두점 또는 새행을 포함하지 않는것으로 가정한다.

여기에 saveText()에 의한 실례파일출력이 있다.

```
The False Shepherds:Hans Bol:1576
Panoramic Landscape:Jan Brueghel the Younger:1619
Dune Landscape:Jan van Goyen:1630
River Delta:Jan van Goyen:1653
```

그러면 파일에서 자료를 읽는 방법을 고찰하자.

```
bool Gallery::loadText(const QString &fileName)
{
```

```
    QFile file(fileName);
    if (!file.open(IO_ReadOnly | IO_Translate)) {
        ioError(file, tr("Cannot open file %1 for reading"));
        return false;
    }
    drawings.clear();
    QTextStream in(&file);
    in.setEncoding(QTextStream::UnicodeUTF8);
    while (!in.atEnd()) {
        Drawing drawing;
        in >> drawing;
        drawings.push_back(drawing);
    }
    if (file.status() != IO_Ok) {
        ioError(file, tr("Error reading from file %1"));
        return false;
    }
    return true;
}
```

흥미있는 부분은 while순환이다. 유효자료가 있는동안 >>연산자에 의하여 읽어들인다. >>연산자의 실현은 그리 간단하지 않다. 다음의 실례를 고찰하자.

```
out << "alpha" << "bravo";
```

out가 QTextStream이면 실제로 써넣어지는 자료는 문자렬 "alphabravo"이다. QTextStream에서 읽어들일 때 실제로 이것을 기대할수 없다.

```
in >> str1 >> str2;
```

사실상 그때 발생하는 일은 str1의 전체 단어 "alphabravo"를 얻고 str2은 아무것도 없다. QDataStream은 문자자료의 앞에 매개 문자렬의 길이를 보관하므로 문제가 없다.

써넣은 본문이 하나의 단어로 이루어지면 그것들사이에 공백을 넣고 단어별로 자료를 읽을수 있다. (8장의 DiagramView::copy()와 DiagramView::paste()함수는 이 수법을 사용한다.) 저자의 이름과 그림의 제목은 보통 1개이상의 단어를 포함하므로 그림의 경우에 이것을 수행할수 없다. 그리하여 매개 행을 전부 읽어들이고 그것을 QStringList::split()에 의해 마당들로 분리한다.

```
QTextStream &operator>>(QTextStream &in, Drawing &drawing)
{
    QString str = in.readLine();
```

```
    QStringList fields = QStringList::split(":", str);
    if (fields.size() == 3) {
        drawing.myTitle = fields[0];
        drawing.myArtist = fields[1];
        drawing.myYear = fields[2].toInt();
    }
    return in;
}
```

QTextStream::read()를 사용하여 한번에 전체 본문을 읽을수 있다.

```
QString wholeFile = in.read();
```

결과문자렬에서 매개 행의 끝은 읽어들이는 파일에 의해 사용되는 행마감관례에 관계없이 새행문자('\n')로 지정된다.

전체 본문파일의 읽기는 자료를 앞처리하여야 하는 경우에 아주 편리하다. 례를 들면

```
wholeFile.replace("&", "&");
wholeFile.replace("<", "<");
wholeFile.replace(">", ">");
```

한번에 써넣기 위하여 자료를 모두 하나의 문자렬에 넣고 그것을 한번에 출력한다.

```
QString Gallery::saveToString()
{
    QString result;
    QTextOStream out(&result);
    list<Drawing>::const_iterator it = drawings.begin();
    while (it != drawings.end()) {
        out << *it;
        ++it;
    }
    return result;
}
```

본문을 파일에 출력하는것처럼 문자렬에 본문을 출력하는것은 간단하며 역시 <<연산자에 기초한다.

```
void Gallery::readFromString(const QString &data)
{
    QString string = data;
    drawings.clear();
    QTextIStream in(&string);
```

```
    while (!in.atEnd()) {
        Drawing drawing;
        in >> drawing;
        drawings.push_back(drawing);
    }
}
```

QTextStream에 의하여 문자렬에서 자료를 발취하는것은 간단하다. >>연산자에 의거하므로 문장해석은 필요없다.

본문자료의 써넣기는 어렵지 않지만 본문읽기는 잘 되지 않는다. 복잡한 형식에 대해서는 완전확장된 문장해석기가 필요하다. 일반적으로 그러한 문장해석기는 QChar에 대하여 >>에 의해 한 문자씩 자료를 읽어들이거나 readLine()에 의하여 한 행씩 자료를 읽어들이고 돌아온 QString을 순환하면서 작업한다.

제3절. 파일과 등록부의 조종

Qt의 QDir클라스는 등록부를 항행하면서 파일들에 대한 정보를 얻기 위한 가동환경에 의존하지 않는 수단을 제공한다. QDir의 사용방법을 알기 위하여 특정한 등록부와 임의의 깊이의 모든 보조등록부들에서 모든 화상들에 의해 소비되는 공간을 계산하는 자그마한 콘솔응용프로그람을 작성한다.

응용프로그람의 핵심부는 imageSpace()함수로서 주어진 등록부의 크기를 계산한다.

```
int imageSpace(const QString &path)
{
    QDir dir(path);
    QStringList::Iterator it;
    int size = 0;
    QStringList files = dir.entryList("*.png *.jpg *.jpeg", QDir::Files);
    it = files.begin();
    while (it != files.end()) {
        size += QFileInfo(path, *it).size();
        ++it;
    }
    QStringList dirs = dir.entryList(QDir::Dirs);
    it = dirs.begin();
    while (it != dirs.end()) {
        if (*it != "." && *it != "..")
            size += imageSpace(path + "/" + *it);
        ++it;
```

```
        }
        return size;
    }
```

주어진 경로를 리용하여 QDir객체를 창조하는것으로 시작한다. entryList()함수에 2개의 인수를 넘긴다. 첫째 인수는 공백으로 구분한 파일이름려과기들의 목록이다. 이것들은 대리기호 '*'와 '?'들을 포함한다. 이 실례에서는 오직 PNG와 JPEG파일들만 포함하도록 려과한다. 둘째 인수는 어떤 종류의 항목(보통파일, 등록부, 구동기 등.)들을 포함하는가 지정한다.

파일목록을 순환하면서 그 크기를 루계한다. QFileInfo클라스는 파일의 크기, 허가, 소유자, 시간과 같은 속성들을 호출하게 한다.

두번째의 entryList()호출에서 이 등록부의 모든 보조등록부들을 얻고 그것들을 항행하면서 imageSpace()을 재귀호출하여 루계화상크기를 확인한다.

매개 보조등록부의 경로를 창조하기 위해서는 현재등록부의 경로를 사선으로 구분하여 보조등록부이름(*it)과 결합한다. QDir는 '/'를 모든 가동환경에서 등록부구분기호로서, Windows에서는 '\'를 구분기호로서 취급한다. 사용자에게 경로를 표시할 때 정적함수 QDir::convertSeparators()를 호출하여 사선들을 가동환경에 고유한 정확한 구분기호로 변환한다.

프로그람에 main()함수를 추가한다.

```
    int main(int argc, char *argv[])
    {
        QString path = QDir::currentDirPath();
        if (argc > 1)
            path = argv[1];
        cerr << "Space used by images in " << endl
            << path.ascii() << endl
            << "and its subdirectories is"
            << (imageSpace(path) / 1024) << " KB" << endl;
        return 0;
    }
```

이 실례에서는 Qt의 도구클라스들만 사용하므로 QApplication객체를 요구하지 않는다.

QDir::currentDirPath()를 사용하여 경로를 현재등록부로 초기화한다. 또한 QDir::homeDirPath()를 리용하여 경로를 사용자의 홈등록부로 초기화할수 있다. 사용자가 지령행에 경로를 지정하면 그대신 그것을 사용한다. 끝으로 imageSpace()함수에 의하여 화상들이 소비한 공간을 계산한다.

QDir클라스는 rename(), exists(), mkdir(), rmdir()를 비롯한 다른 파일들과 등록부관련함수들을 제공한다. QFile클라스는 remove()와 exists()를 비롯한 정적편의함수들을 제공한다.

제4절 프로쎄스사이통신

QProcess클라스는 외부프로그람들을 실행하고 교제하게 한다. 이 클라스는 비동기적으로 작업하며 사용자대면부가 응답성을 유지하도록 배경에서 자기 작업을 수행한다. QProcess는 외부프로쎄스가 자료를 가지거나 완료했을 때 통지하기 위한 신호들을 발생한다.

외부화상편의프로그람의 사용자대면부를 제공하는 자그마한 응용프로그람을 개발한다. 이 실례에서는 모든 주요 가동환경들에서 자유로 사용할수 있는 ImageMagick변환프로그람을 사용하게 한다.

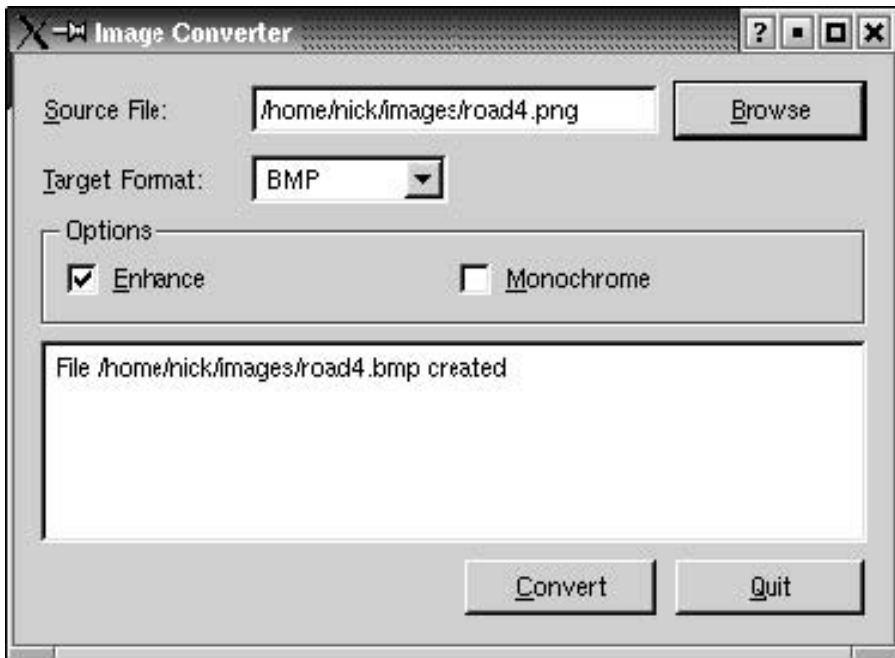



그림 10-1. Image Converter응용프로그람

Image Converter의 사용자대면부는 Qt Designer로 창조한다. 여기서는 코드를 포함하는 .ui.h 파일에 초점을 둔다. process와 fileFilters변수들은 Qt Designer의 Members타브에서 다음과 같이 선언되여있다.

```
QProcess *process;QString fileFilters;
```

uic도구는 이 변수들을 ConvertDialog클라스의 부분으로 포함한다.

```
void ConvertDialog::init()
{
    process = 0;
    QStringList imageFormats = QImage::outputFormatList();
    targetFormatComboBox->insertStringList(imageFormats);
    fileFilters = tr("Images") + " (*." + imageFormats.join(" *.").lower() + ")";
}
```

파일려과기는 설명본문과 하나이상의 대리기호견본들로 구성된다. (례를 들면 "Text

files(*.txt)"). QImage::outputFormatList()함수는 Qt가 유지하는 화상출력형식들의 목록을 돌려준다. 이 목록은 Qt가 설치될 때 설정된 선택들에 따라 달라질수 있다.

```
void ConvertDialog::browse()
{
    QString initialName = sourceFileEdit->text();
    if (initialName.isEmpty())
       initialName = QDir::homeDirPath();
    QString fileName = QFileDialog::getOpenFileName(initialName, fileFilters, this);
    fileName = QDir::convertSeparators(fileName);
    if (!fileName.isEmpty()) {
       sourceFileEdit->setText(fileName);
       convertButton->setEnabled(true);
    }
}
```

대화칸의 Browse단추는 browse()처리부에 련결된다. 사용자가 이미 파일을 선택하였다면 파일대화칸을 그 파일의 경로로 초기화하고 그렇지 않으면 사용자의 홈등록부를 사용한다.

```
void ConvertDialog::convert()
{
    QString sourceFile = sourceFileEdit->text();
    targetFile = QFileInfo(sourceFile).dirPath() + QDir::separator() +
                             QFileInfo(sourceFile).baseName();
    targetFile += ".";
    targetFile += targetFormatComboBox->currentText().lower();
    convertButton->setEnabled(false);
    outputTextEdit->clear();
    process = new QProcess(this);
    process->addArgument("convert");
    if (enhanceCheckBox->isChecked())
         process->addArgument("-enhance");
    if (monochromeCheckBox->isChecked())
         process->addArgument("-monochrome");
    process->addArgument(sourceFile);
    process->addArgument(targetFile);
    connect(process, SIGNAL(readyReadStderr()), this, SLOT(updateOutputTextEdit()));
    connect(process, SIGNAL(processExited()), this, SLOT(processExited()));
```

```
        process->start();
    }
```

대화칸의 Convert단추는 convert()처리부에 련결된다. 원천파일의 이름을 복사하고 목표파일형식과 일치하도록 그 뒤붙이를 변경한다.

그다음 QProcess객체를 창조한다. addArgument()을 사용하는 QProcess객체에 주어진 첫째 인수는 실행하려는 외부프로그람의 이름이다. 그 뒤의 인수들은 이 프로그람의 인수로 된다.

QProcess의readyReadStderr()를 대화칸의 updateOutputTextEdit()처리부에 련결하여 오유가 발생하였을 때 외부프로그람으로부터 대화칸의 QTextEdit에 오유통보문들을 현시한다. 또한 QProcess의 precessExited()신호를 대화칸의 같은 이름의 처리부에 련결한다.

```
void ConvertDialog::updateOutputTextEdit()
{
    QByteArray data = process->readStderr();
    QString text = outputTextEdit->text() + QString(data);
    outputTextEdit->setText(text);
}
```

외부프로쎄스가 stderr에 써넣을 때마다 updateOutputTextEdit()처리부가 호출된다. 그 오유본문을 읽어서 QTextEdit에 추가한다.

```
void ConvertDialog::processExited()
{
    if (process->normalExit()) {
        outputTextEdit->append(tr("File %1 created") .arg(targetFile));
    } else {
        outputTextEdit->append(tr("Conversion failed"));
    }
    delete process;
    process = 0;
    convertButton->setEnabled(true);
}
```

프로쎄스가 완료되였을 때 사용자에게 결과를 알리고 프로쎄스를 삭제한다.

이와 같이 콘솔응용프로그람을 포함하는 방법은 현존기능을 다시 실현하지 않고 그 기능자체를 사용하게 하므로 더 효과적이다. QProcess의 다른 사용은 웨브열람기나 email의뢰기와 같은 다른 GUI응용프로그람들을 호출하는것이다.

제11장. 용기클라스

용기클라스는 주어진 형의 항목들을 기억기에 보관하는 일반목적형판클라스이다. 표준 C++는 이미 표준형판서고(STL)의 일부로서 많은 용기들을 포함하고있다.

Qt는 자체의 용기클라스들을 제공하므로 Qt프로그람을 쓸 때 Qt와 STL의 용기를 둘다 사용할수 있다. 자기가 이미 STL용기에 습관되여있고 자기의 목표가동환경에 유효한 STL이 있으면 Qt용기를 리용할 특별한 리유란 없다.

이 장에서는 대부분의 중요한 STL과 Qt용기들을 개괄한다. 또한 QString과 QVariant를 고찰한다. 두 클라스들은 용기들과 공통점이 많고 일부 상황에서는 용기대신에 사용할수 있다.

제1절. 벡토르

벡토르, 목록, 매프용 STL과 Qt클라스들은 보관하려는 객체들의 형의 따라 파라메터화된 형판클라스들이다. 이 클라스들에 보관할수 있는 값들은 기본형(int와 double), 지적자, 혹은 기정구성자(인수없는 구성자), 복사구성자, 대입연산자를 가지는 클라스일수 있다. 수식하는 클라스들에는 QDateTime, QRegExp, QString, QVariant들이 포함된다. QObject를 계승하는 Qt클라스들은 수식하지 않는다. 그것은 이 클라스들이 복사구성자와 대입연산자를 실현하지 않기때문이다. 이것은 보통 문제가 아니므로 여전히 이러한 형들에 지적자들을 보관할수 있다.

이 절에서는 벡토르에 대한 가장 일반적인 조작들을 고찰하고 다음의 두개 절에서는 목록과 매프를 개괄한다. 대부분의 실례들에서는 영화제목과 재생시간을 보관하는 Film클라스를 사용한다. (동화상을 표시하는데 쓰이는 Qt의 QMovie클라스와 너무나도 비슷하므로 Movie클라스라고 부르지 않는다.)

여기에 Film의 정의가 있다.

```
class Film
{
public:
    Film(int id = 0, const QString &title = "", int duration = 0);
    int id() const { return myId;}
    void setId(int catalogId) { myId = catalogId;}
    QString title() const { return myTitle;}
    void setTitle(const QString &title) { myTitle = title;}
    int duration() const { return myDuration;}
    void setDuration(int minutes) { myDuration = minutes;}
private:
    int myId;
    QString myTitle;
```

```
    int myDuration;
};
int operator==(const Film &film1, const Film &film2);
int operator<(const Film &film1, const Film &film2);
```

C++가 자동적으로 제공하는것이면 충분하므로 복사구성자나 대입연산자를 명시적으로 제공하지 않는다. 클라스가 그 클라스에 의해 할당된 기억기의 지적자들을 포함한다면 자체로 지적자들을 실현해야 한다.

클라스와 함께 같기연산자와 작기연산자를 제공한다. 같기연산자는 용기에 특정한 항목이 있는가 탐색할 때 쓰인다. 작기연산자는 항목들을 정렬하기 위하여 비교하는데 쓰인다. 4개의 다른 비교연산자들(!=, <=, >, >=)은 STL이 절대로 사용하지 않으므로 실현할 필요가 없다.

여기에 3개의 비inline함수들의 실현이 있다.

```
Film::Film(int id, const QString &title, int duration)
{
    myId = id;
    myTitle = title;
    myDuration = duration;
}
int operator==(const Film &film1, const Film &film2)
{
    return film1.id() == film2.id();
}
int operator<(const Film &film1, const Film &film2)
{
    return film1.id() < film2.id();
}
```

Film객체들을 비교할 때 제목들이 절대로 유일하지 않으므로 제목보다 ID들을 사용한다.

Film(4812 "A Hard Day's Night", 85)	films[0]
Film(5051 "Seven Days to Noon", 94)	films[1]
Film(1301 "Day of Wrath", 105)	films[2]
Film(9227 "A Special Day", 110)	films[3]
Film(1817 "Day for Night", 116)	films[4]

그림 11-1. Film들의 벡토르

벡토르는 기억기의 이웃위치들에 항목들을 보관하는 자료구조이다. 일반C++배렬과 벡토르의 차이는 벡토르는 자기의 크기를 알고있고 크기를 변경할수 있다는것이다. 벡토르의 끝에 여분의 요소들을 추가하는것은 아주 효과있으나 벡토르의 앞이나 중간에 요소들을 삽입하는것은 품이 든다.

STL의 vector클라스는 std::vector<T>이고 <vector>에서 정의된다. 여기에 실례가 있다.

```
vector<Film> films;
```

Qt에서 벡토르는 QValueVector<T>이다.

```
QValueVector<Film> films;
```

이렇게 창조한 벡토르는 크기가 0이다. 몇개의 요소가 필요한가를 미리 알고있으면 벡토르를 정의할 때 초기값을 줄수 있고 []연산자를 리용하여 요소들에 값을 대입하며 그렇지 않으면 후에 크기를 조절하거나 항목들을 추가해야 한다.

벡토르를 채우는 편리한 수법은 push_back()를 사용하는것이다. 이 함수는 벡토르를 하나씩 확장하면서 끝에 요소를 하나씩 추가한다.

```
films.push_back(Film(4812, "A Hard Day's Night", 85));
films.push_back(Film(5051, "Seven Days to Noon", 94));
films.push_back(Film(1301, "Day of Wrath", 105));
films.push_back(Film(9227, "A Special Day", 110));
films.push_back(Film(1817, "Day for Night", 116));
```

일반적으로 Qt는 STL과 같은 함수이름들을 제공하지만 일부 경우에 Qt는 Qt풍의 이름들을 추가적으로 가진다. 례를 들면 Qt클라스들을 사용하면 push_back()나 append()를 리용하여 항목들을 추가할수 있다.

벡토르를 채우는 다른 방법은 벡토르에 초기크기를 주고 요소들을 개별적으로 초기화하는것이다.

```
vector<Film>  films(5);
films[0] = Film(4812, "A Hard Day's Night", 85);
films[1] = Film(5051, "Seven Days to Noon", 94);
films[2] = Film(1301, "Day of Wrath", 105);
films[3] = Film(9227, "A Special Day", 110);
films[4] = Film(1817, "Day for Night", 116);
```

명시적인 값을 주지 않고 창조한 벡토르항목들은 항목클라스의 기정구성자에 의해 초기화된다. 기본형과 지적자형들에 대하여 마치도 탄창에 이러한 형의 변수들을 정의할 때처럼 값이 정의되지 않는다.

[]연산자를 리용하여 벡토르의 요소들을 순환할수 있다.

```
for (int i = 0;i < (int)films.size();++i)
    cerr << films[i].title().ascii() << endl;
```

또한 반복자를 사용할수 있다.

```
vector<Film>::const_iterator it = films.begin();
while (it != films.end()) {
    cerr << (*it).title().ascii() << endl;
    ++it;
}
```

매개 용기클라스는 2개의 반복자형 iterator와 const_iterator를 가지고있다. 두 반복자사이의 차이는 const_iterator가 자료를 수정할수 없게 한다는것이다.

용기의 begin()함수는 용기의 첫 항목(례를 들면 films[0])을 참고하는 반복자를 돌려준다. 용기의 end()함수는 마지막에서 하나 전 항목(례를 들면, films[5])을 참고하는 반복자를 돌려준다. 용기가 비였으면 begin()은 end()와 같다. 이것은 용기가 어떤 요소를 가지는가 확인하는데 쓰일수 있다. 물론 이러한 목적에는 empty()를 호출하는것이 더 편리하다.

반복자는 C++지적자와 비슷한 문법을 따른다. ++와 –연산자들을 사용하여 다음 항목이나 이전 항목으로 이동할수 있고 단항*연산자를 리용하여 현재반복자위치에 보관된 항목을 얻을수 있다. 사실상 vector<T>에서 iterator와 const_iterator형들은 순수 T *와 const T *의 형정의이다.

선형탐색으로 벡토르의 항목을 찾으려고 한다면 STL find()함수를 사용할수 있다.

```
vector<Film>::iterator it = find(films.begin(), films.end(), Film(4812));
if (it != films.end())
    films.erase(it);
```

find()함수는 마지막 인수로서 넘긴 항목과 같은(연산자==()리용) 첫 항목에로의 반복자를 돌려준다. 이것은 <algorithm>에서 다른 많은 형판함수들과 함께 정의된다. 이 함수들은 일반적으로 반복자들에 대하여 조작한다. Qt는 각이한 이름들(례를 들면 qFind())을 가지는 함수들을 일부 제공한다. STL 없이 Qt를 사용하려고 한다면 이러한 함수들을 사용할수 있다.

벡토르에서 항목들을 정렬하기 위하여 sort()를 호출할수 있다.

```
sort(films.begin(), films.end());
```

sort()함수는 다른 비교함수가 넘어오지 않으면 <연산자를 사용하여 항목들을 비교한다.

정렬되면 binary_search()함수에 의하여 항목이 존재하는가 확인한다. 정렬된 벡토르에서 binary_search()는 find()와 같은 결과를 주지만(2개의 영화가 같은 ID를 가지지 않는다고 가정) 훨씬 빠르다.

```
int id = 1817;
if (binary_search(films.begin(), films.end(), Film(id)))
    cerr << "Found" << id << endl;
```

반복자가 가리키는 위치가 주어지면 품을 들여 insert()를 리용하여 새로운 항목을 삽입하거나 erase()에 의해 현존항목을 삭제할수 있다.

```
films.erase(it);
```

벡토르에서 삭제된 항목뒤에 오는 항목들은 그때 왼쪽으로 한 위치 이동되여 그 위치를 채우고 벡토르의 크기는 하나 줄어든다.

제2절. 목록

목록(혹은 련결목록)은 기억기의 이웃하지 않은 위치들에 항목들을 보관하는 자료구조이다. 벡토르와 달리 목록은 아주 빈약한 직접호출기능을 가지며 다른 한편 삽입과 삭제는 아주 빠르다.

벡토르에서 동작하는 대부분의 알고리듬(특히 sort()와 binary_search())은 목록에서 동작하지 않는다. 이것은 목록이 고속직접호출을 제공하지 않기때문이다. STL목록을 정렬하는데 sort()성원함수를 리용한다.

Film(4812, "A Hard Day's Night", 85)   films.begin()
Film(5051, "Seven Days to Noon", 94)
Film(1301, "Day of Wrath", 105)
Film(9227, "A Special Day", 110)
Film(1817, "Day for Night", 116)
films.end()

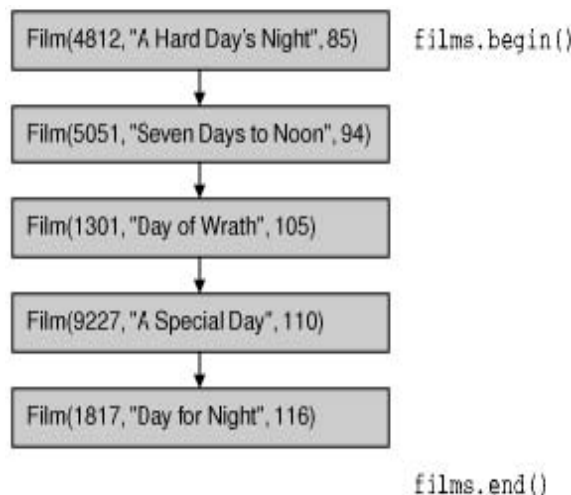



그림 11-2. Film들의 목록

STL의 목록클라스는 std::list<T>이고 <list>에서 정의된다. 여기에 실례가 있다.

```
list<Film> films;
```

Qt에서 목록은 QValueList<T>이다.

```
QValueList<Film> films;
```

Film클라스는 앞절에서 제시하였다.

새 항목들은 push_back()나 insert()에 의해 추가된다. 벡토르와 달리 목록의 선두나 중간에서 삽입은 품이 들지 않는다.

STL목록은 []연산자를 제공하지 않으므로 요소들을 항행하는데 반복자를 사용해야 한다. (Qt목록은 []연산자를 유지하지만 큰 목록에서 아주 느릴수 있다.) 문법과 사용법은 벡토르와 같은데 반복자형의 앞에 vector<T>라고 쓰지 않고 list<T>라고 쓴다. 례를 들면

```
list<Film>::const_iterator it = films.begin();
while (it != films.end()) {
   cerr << (*it).title().ascii() << endl;
```

```
        ++it;
    }
```

또한 목록은 대체로 empty(), size(), erase(), clear()를 비롯하여 벡토르와 같은 함수들을 제공한다. find()알고리듬도 목록에서 사용할수 있다.

일부 Qt함수들은 QValueList<T>를 돌려준다. 함수의 돌림값에 대하여 순환하려면 목록의 사본을 얻고 사본에 대하여 순환해야 한다. 례를 들면 다음의 코드는 QSplitter::sizes()가 돌려준 QValueList<int>를 순환하는 정확한 방법이다.

```
    QValueList<int> list = splitter->sizes();
    QValueList<int>::const_iterator it = list.begin();
    while (it != list.end()) {
        do_something(*it);
        ++it;
    }
```

다음의 코드는 틀린다.

```
    // WRONG
    QValueList<int>::const_iterator it = splitter->sizes().begin();
    while (it != splitter->sizes().end()) {
        do_something(*it);
        ++it;
    }
```

이것은 QSplitter::sizes()가 호출될 때마다 값에 의해 새로운 QValueList<int>를 돌려주기때문이다. 돌림값을 보관하지 않으면 C++는 순환을 시작하기전에 자동적으로 돌림값을 해체하므로 매달려있는 반복자를 남기게 된다. 또한 매번 순환할 때마다 QSplitter::sizes()가 splitter->sizes().end()호출로 인하여 목록의 새로운 사본을 생성해야 하는것이다.

요약: 늘 값에 의해 되돌아오는 용기의 사본에 대하여 순환한다.

이러한 용기의 복사는 품이 든다고 하지만 그렇지 않다. 그것은 Qt가 암시적공유(implicit sharing)라는 최적화를 리용하기때문이다. 이 최적화는 지어는 배경에서 자료복사가 발생하지 않아도 자료를 복사하는것처럼 프로그람을 작성할수 있다는것을 의미한다.

Qt에서 많이 쓰이는 QStringList클라스는 QValueList<QString>의 파생클라스이다. 이 클라스는 그 기초클라스로부터 계승하는 함수들과 함께 클라스를 더 강력하게 하는 여분의 함수들을 제공한다. 이러한 함수들은 이 장의 마지막 절에서 론의한다.

제3절. 매프

매프는 건에 의해 첨수화된 같은 형의 항목들을 임의의 개수 보관한다. 매프는 건마다 하나의 값을 보관한다. 매프는 우수한 직접호출과 삽입기능을 가진다. 새 값이 현존 건에 할당되면 낡은 값은 새 값으로 교체된다.

1301	Film("Day of Wrath", 105)	films[1301]
1817	Film("Day for Night", 116)	films[1817]
4812	Film("A Hard Day's Night", 85)	films[4812]
5051	Film("Seven Days to Noon", 94)	films[5051]
9227	Film("A Special Day", 110)	films[9227]

그림 11-3. Film들의 매프

매프가 건–값쌍들을 포함하므로 일반적으로 벡토르와 목록의 설계와는 현저히 다른 방법으로 매프와 작업하는 자료구조를 설계한다. 여기에 매프의 사용법을 설명하는데 쓰이는 Film 클라스의 판이 있다.

```
class Film
{
public:
    Film(const QString &title = "", int duration = 0);
    QString title() const { return myTitle;}
    void setTitle(const QString &title) { myTitle = title;}
    int duration() const { return myDuration;}
    void setDuration(int minutes) { myDuration = minutes;}
private:
    QString myTitle;int myDuration;
};
Film::Film(const QString &title, int duration)
{
    myTitle = title;
    myDuration = duration;
}
```

Film클라스에서는 일람표ID를 매프의 건으로 사용하므로 그 ID를 보관하지 않는다. Film에는 비교연산자들이 필요없다. 매프는 값이 아니라 건에 의하여 순서화된다.

STL의 map클라스는 std::map<K, T>이고 <map>에서 정의된다. 여기에 건이 int(일람표ID)이고 그 값이 Film인 매프의 실례가 있다.

```
map<int, Film> films;
```

Qt의 매프는 QMap<K, T>이다.

```
QMap<int, Film> films;
```

매프에 항목들을 삽입하는 가장 원시적인 수법은 주어진 건에 값을 할당하는것이다.

```
films[4812] = Film("A Hard Day's Night", 85);
films[5051] = Film("Seven Days to Noon", 94);
films[1301] = Film("Day of Wrath", 105);
films[9227] = Film("A Special Day", 110);
films[1817] = Film("Day for Night", 116);
```

매프반복자는 건–값쌍을 제공한다. 건부분은 (*it).first에 의해 값부분은 (*it).second에 의해 꺼낸다.

```
map<int, Film>::const_iterator it = films.begin();
while (it != films.end()) {
    cerr << (*it).first << ": " << (*it).second.title().ascii() << endl;
    ++it;
}
```

또한 대부분의 콤파일러들은 it->first와 it->second를 쓸수 있게 하지만 (*it).first와 (*it).second라고 쓰는것이 더 이식성있다.

Qt매프의 반복자는 STL매프의 반복자와 현저히 다르다. Qt매프에서 건은 it.key()에 의하여, 값은 it.data()에 의하여 반복자로부터 얻어진다.

```
QMap<int, Film>::const_iterator it = films.begin();
while (it != films.end()) {
    cerr << it.key() << ": " << it.data().title().ascii() << endl;
    ++it;
}
```

매프를 순환할 때 항목들은 늘 건에 의해 순서화된다.

[]연산자를 삽입과 얻기에 모두 사용할수 있으나 []으로 존재하지 않는 건의 값을 얻으려고 한다면 새 항목이 주어진 건과 빈 값으로 창조된다. 우연히 빈 값을 창조하지 않게 하려면 find()성원함수에 의하여 항목들을 얻는다.

```
map<int, Film>::const_iterator it = films.find(1817);
if (it != films.end())
    cerr << "Found " << (*it).second.title().ascii() << endl;
```

이 함수는 건이 매프에 없으면 end()반복자를 돌려준다.

실례에서는 옹근수건을 사용하였으나 다른 형의 건들도 가능하며 한가지 일반적인 선택은 QString건이다. 례를 들면

```
map<QString, QString> actorToNationality;
actorToNationality["Doris Day"] = "American";
actorToNationality["Greta Garbo"] = "Swedish";
```

같은 건에 대하여 여러개의 값을 보관하려고 한다면 multimap<K, T>를 사용할수 있다. 오직 건을 보관해야 한다면 set<K> 혹은 multiset<K>를 사용한다. Qt는 이것과 등가한 클래스들을 제공한다.

Qt의 QMap<K, T>클라스는 작은 자료모임을 취급할 때 특별히 쓸모있는 추가적인 편의함수들을 가지고있다. QMap<K, T>::keys()와 QMap<K, T>::values()는 매프의 건과 값들의 QValueList들을 돌려준다.

제4절. 지적자에 기초하는 용기

앞절에서 서술한 STL용기와 같이 Qt도 용기클라스들의 추가모임을 제공한다. 이러한 클라스들은 STL이 C++의 부분으로 되기전인 1990년대초에 Qt 1.0용으로 개발되였으므로 자기의 특수한 문법을 가진다. 이 클라스들이 객체의 지적자들에 대하여 조작하므로 흔히 지적자에 기초한 용기로서 참고되며 Qt와 STL의 값기초용기들과 대조된다. Qt 4에서 지적자에 기초한 용기들은 호환성으로 하여 계속 사용하게 되겠지만 값에 기초한 용기들에 비하여 널리 사용되지 않으리라고 기대한다.

지적자에 기초한 클라스들을 새로 쓴 Qt코드에서 사용하는 주요한 원인은 Qt 3의 일부 중요함수들이 거기에 기초하고있기때문이다. 이미 3장에서 응용프로그람의 제일 웃준위창문부품을 순환하는 실례를 보았으며 6장에서 응용프로그람의 MDI창문들을 순환하는 다른 실례를 보았다.

지적자에 기초하는 주요한 용기들은 QPtrVector<T>, QPtrList<T>, QDict<T>, QAsciiDict<T>, QIntDict<T>, 그리고 QPtrDict<T>이다.

QPtrVector<T>는 지적자들의 벡토르를 보관한다. 여기에 5개의 Film객체들을 QPtrVector<Film>에 옮기는 실례가 있다.

```
QPtrVector<Film> films(5);
films.setAutoDelete(true);
films.insert(0, new Film(4812, "A Hard Day's Night", 85));
films.insert(1, new Film(5051, "Seven Days to Noon", 94));
films.insert(2, new Film(1301, "Day of Wrath", 105));
films.insert(3, new Film(9227, "A Special Day", 110));
films.insert(4, new Film(1817, "Day for Night", 116));
```

QPtrVector<T>는 append()함수를 제공하지 않으므로 벡토르의 크기를 자체로 조절해야 하며 항목들을 특정한 첨수위치에 삽입한다.이 실례에서는 일람표ID들을 포함하는 원래의 Film클라스를 사용한다.

Qt의 지적자에 기초한 용기들의 한가지 우월한 특성은 《자동삭제》속성이다. 자동삭제가 허용되면 Qt는 용기에 모든 객체들을 삽입할 권한을 가지며 용기가 삭제될 때(혹은 remove()나 clear()가 사용될 때) 객체들을 삭제한다.

벡트로에서 항목을 삭제하기 위하여 첨수를 넘기여 remove()를 호출할수 있다.

```
films.remove(2);
```

remove()조작은 벡토르의 크기를 변경하지 않으며 그대신에 항목은 null지적자로 설정된다. 자동삭제가 허용되면 항목은 자동적으로 삭제된다.

QPtrVector<T>를 항행하기 위하여서는 단순히 첨수를 사용할수 있다.

```
for (int i = 0;i < (int)films.count();++i) {
    if (films[i])
        cerr << films[i]->title().ascii() << endl;
}
```

주어진 첨수의 지적자를 사용하기전에 그것이 null이 아닌가 검사하며 그러한 경우에는 삭제되였거나 거기에 할당된것이 없다.

QPtrList<T>클라스는 지적자들의 목록을 보관한다. append(), prepend(), 혹은 insert()를 호출하여 QPtrList<T>에 새 항목들을 추가할수 있다.

```
QPtrList<Film> films;
films.setAutoDelete(true);
films.append(new Film(4812, "A Hard Day's Night", 85));
films.append(new Film(5051, "Seven Days to Noon", 94));
```

지적자목록은 first(), next(), prev(), last()와 같은 항행함수들을 호출할 때 갱신되는 《현재》항목을 가지고있다. 목록을 순환하는 한가지 수법은 다음과 같다.

```
Film *film = films.first();
while (film) {
  cerr << film->title().ascii() << endl;
  film = films.next();
}
```

또한 at()에 의해 목록을 순환할수도 있다.

```
for (int i = 0; i < (int)films.count(); ++i)
    cerr << films.at(i)->title().ascii() << endl;
```

셋째 선택은 QPtrListIterator<T>를 사용하는것이다.

QDict<T>, QAsciiDict<T>, QIntDict<T>, QPtrDict<T>클라스들은 map<K, T>와 제일 가까운 지적자에 기초한 클라스들이다. 이러한 클라스들은 또한 건–값쌍들에도 조작한다. 건은 4개 클라스들중 어느것을 사용하는가에 따라서 4개의 다른 형(QString, const char *, int, 혹은 void *)중 하나일수 있다. 4개 클라스 모두가 같은 함수들을 제공하므로 QIntDict<T>를 고찰한다.

이 클라스를 사용하여 처음에 map<K, T>에서 사용한것과 같은 형의 Film들을, 일람표ID를 건으로 리용하여 보관한다.

```
QIntDict<Film> films(101);
films.setAutoDelete(true);
```

QIntDict<T>구성자는 수를 받아들인다. 그 수는 자료를 넣는 용기가 몇개인가를 결정하기 위하여 클라스가 내적으로 사용한다. 성능을 높이기 위하여 그 수는 보관하려는 항목수보다 좀 큰 씨수로 되여야 한다.

새로운 항목들은 건과 값을 받아들이는 insert()로 삽입한다.

```
films.insert(4812, new Film("A Hard Day's Night", 85));
films.insert(5051, new Film("Seven Days to Noon", 94));
```

find()나 []연산자에 의하여 항목들을 얻고 remove()에 의하여 항목을 삭제하며 replace()로 주어진 건과 련상된 값을 교체한다.

같은 건으로 insert()를 여러번 호출하면 오직 제일 최근에 삽입된 항목을 호출할수 있다. remove()를 호출하면 항목들은 그것들이 삽입된 순서와는 반대로 삭제된다. 같은 건에 대하여 여러값을 넣는것을 피하기 위하여 insert()대신에 replace()를 사용할수 있다.

반복자를 사용하여 용기전체를 횡단할수 있다.

```
QIntDictIterator<Film> it(films);
while (it.current()) {
    cerr << it.currentKey() << ": " << it.current()->title().ascii() << endl;
    ++it;
}
```

반복자는 currentKey()로 현재건을, current()로 현재값을 제공한다. 항목들이 나타나는 순서는 정의되지 않는다.

Qt는 벡토르류의 특수한 클라스 QMemArray<T>를 제공하는데 이것은 int와 double과 같은 기본형의 항목들이나 기본형들의 구조체들을 보관한다. 일부 응용프로그람들은 그것을 직접 사용하지만 두 파생클라스 QByteArray(QMemArray<char>)와 QPointArray(QMemArray<QPoint>)는 아주 일반적이며 앞 장들에서 여러번 사용하였다.

례를 들면 여기에 QByteArray를 창조하는 방법이 있다.

```
QByteArray bytes(4);
bytes[0] = 'A';
bytes[1] = 'C';
bytes[2] = 'D';
bytes[3] = 'C';
```

QMemArray<T>를 창조할 때 거기에 초기크기를 넘기거나 후에 resize()를 호출할수 있다. 그다음 []연산자에 의하여 배렬항목들을 호출할수 있다.

```
for (int i = 0; i < (int)bytes.size(); ++i)
cerr << bytes[i] << endl;
```

QMemArray<T>::find()을 사용하여 항목을 탐색할수 있다.

```
if (bytes.find('A') != -1)
```

```
cerr << "Found" << endl;
```

QMemArray<T>와 그 파생클라스들에서 포착하기 어려운 결함은 그것들이 명시적공유(explicitly sharing)로 되는것이다. 이것은 클라스의 복사구성자나 대입연산자를 리용하여 객체의 사본을 창조할 때 원본과 사본이 둘다 같은 자료를 공유한다는것을 의미한다. 그것들중 하나를 수정할 때 다른것도 수정된다. 명시적공유를 이러한 문제를 가지지 않는 암시적공유와 혼돈하지 말아야 한다.

QMemArray<T>를 리용하여 프로그람을 짤 때 그 수법은 copy()를 호출하여 그것을 복사할 때 깊은 복사를 하게 하는것이다.

```
duplicate = bytes.copy();
```

이것은 2개의 QMemArray<T>객체들이 같은 자료를 지적하지 않는다는것을 담보한다.

명시적공유에 내재하는 문제들을 피하기 위하여 Qt 4에서는 아마 QMemArray<T>클라스대신에 QValueVector<T>를 사용하게 될것이다. 그때 QByteArray와 QPointArray클라스는 기초클라스로서 QValueVector<T>를 사용할것이다.

제5절. QString과 QVariant

문자렬은 모든 GUI프로그람의 사용자대면부뿐아니라 자료구조로서도 사용된다.

C++는 본래 두 종류의 문자렬 즉 전통적인 C형식의 '\0'완료문자배렬과 string클라스를 제공한다. Qt의 QString클라스는 이것들보다 더 강력하다. QString클라스는 16bit 유니코드값들을 보관한다. 유니코드는 ASCII와 Latin-1을 부분모임으로 포함하며 보통의 수값들로 이루어진다. 그러나 QString이 16bit이므로 세계의 대부분의 언어를 쓰기 위한 수천개의 각이한 문자들을 표시할수 있다.(유니코드에 대한 자세한 정보를 알려면 15장을 참고하시오.)

QString은 2개의 문자렬을 결합하는 2항 +연산자와 한 문자렬을 다른 문자렬에 추가하는 +=연산자를 제공한다. 여기에 실례가 있다.

```
QString str = "User: ";
str += userName + "\n";
```

또한 같은 일을 +=연산자로 수행하는 QString::append()함수도 있다.

```
str = "User: ";
str.append(userName);
str.append("\n");
```

문자렬들을 결합하는 완전히 다른 수법은 QString의 sprintf()함수를 사용하는것이다.

```
str.sprintf("%s %.1f%%", "perfect competition", 100.0);
```

이 함수는 C++서고의 sprintf()함수와 같은 서식지정자를 유지한다. 우의 실례에서 str는 "perfect competition 100.0%"로 된다.

문자렬들이나 수값들로부터 문자렬을 건설하는 또 다른 수법은 arg()를 사용하는것이다.

```
str = QString("%1 %2 (%3s-%4s)") .arg("permissive") .arg("society") .arg(1950) .arg(1970);
```

이 실례에서 "%1"은 "permissive"로, "%2"는 "society"로, "%3"은 "1950"으로, "%4"는 "1970"

으로 교체된다. 결과는 "permissive society (1950s1970s)"이다. 여러가지 자료형들을 처리하기 위한 arg()다중정의가 있다. 일부 다중정의는 마당폭, 밀수 혹은 류점수의 정확도를 조종하기 위한 여분의 파라메터를 가지고있다. 일반적으로 arg()는 형안전하고 유니코드를 완전유지하므로 sprintf()보다 더 좋은 수법이며 번역기들이 *"%n"*파라메터들의 순서를 변경하게 한다.

QString은 QString::number()정적함수에 의하여 수값을 문자렬로 변환할수 있다.

```
str = QString::number(59.6);
```

혹은 setNum()함수를 리용할수 있다.

```
str.setNum(59.6);
```

문자렬로부터 수값에로의 역변환은 toInt(), toLongLong(), toDouble() 등에 의해 수행된다. 례를 들면

```
bool ok;
double d = str.toDouble(&ok);
```

이 함수들은 또한 bool에로의 선택지적자를 받아들이고 변환의 성공여부에 따라 bool을 true나 false로 설정한다. 변환이 실패할 때 이 함수들은 늘 0을 돌려준다.

문자렬이 있으면 흔히 그 일부를 꺼내려고 한다. mid()함수는 주어진 위치로부터 시작하여 주어진 길이를 가지는 부분문자렬을 돌려준다. 례를 들면 다음의 코드는 콘솔에 "pays"를 출력한다.

```
QString str = "polluter pays principle";
cerr << str.mid(9, 4).ascii() << endl;
```

둘째 인수를 생략하면(혹은 –1을 넘기면) mid()는 주어진 위치로부터 시작하여 문자렬의 끝에서 끝나는 부분문자렬을 돌려준다. 례를 들면 다음의 코드는 콘솔에 "pays principle"를 출력한다.

```
QString str = "polluter pays principle";
cerr << str.mid(9).ascii() << endl;
```

또한 비슷한 일을 하는 left()와 right()함수도 있다. 둘다 문자수 *n*을 받아들이고 문자렬의 처음 혹은 마지막 *n*문자를 돌려준다. 례를 들면 다음의 코드는 콘솔에 "polluter principle"을 출력한다.

```
QString str = "polluter pays principle";
cerr << str.left(8).ascii() << " " << str.right(9).ascii() << endl;
```

문자렬이 무엇으로 시작하고 끝나는가를 검사하려고 한다면 startsWith()와 endsWith()함수들을 사용할수 있다.

```
if (uri.startsWith("http:") && uri.endsWith(".png"))
    ...
```

이것은 둘다 다음것보다 간단하고 빠르다.

```
if (uri.left(5) == "http:" && uri.right(4) == ".png")
```

...

==연산자에 의한 문자렬비교는 대소문자를 구별한다. 대소문자를 구별하지 않는 비교인 경우에는 upper()나 lower()를 사용할수 있다. 례를 들면

```
if (fileName.lower() == "readme.txt") ...
```

문자렬의 일정한 부분을 다른 문자렬로 교체하려고 한다면 replace()를 사용할수 있다.

```
QString str = "a sunny day";
str.replace(2, 5, "cloudy");
```

결과는 "a cloudy day"이다. 코드는 remove()와 insert()를 사용하여 다시 쓸수 있다.

```
str.remove(2, 5);
str.insert(2, "cloudy");
```

우선 위치 2로부터 시작하여 5개 문자를 삭제하여 결과문자렬 "a day"(2개의 공백)을 얻은 다음 위치 2에 "cloudy"를 삽입한다.

첫 인수의 모든 출현을 둘째 인수로 교체하는 replace()의 다중정의판이 있다. 례를 들면 여기에 문자렬에서 "&"의 모든 출현을 "&"로 교체하는 방법이 있다.

```
str.replace("&", "&amp;");
```

한가지 빈번히 제기되는 요구는 문자렬에서 공백(공백기호, 타브, 새행 등)을 제거하는것이다. QString은 문자렬의 량 끝으로부터 공백을 제거하는 함수를 가지고있다.

```
QString str = " BOB \t THE \nDOG \n";
cerr << str.stripWhiteSpace().ascii() << endl;
```

문자렬 str는 다음과 같이 표시된다.

			B	O	B		\t		T	H	E			\n	D	O	G		\n

stripWhiteSpace()가 돌려준 문자렬은 다음과 같다.

B	O	B		\t		T	H	E			\n	D	O	G

사용자입력을 처리할 때 흔히 공백을 제거하는것과 함께 하나이상의 내부공백문자의 모든 렬을 하나의 공간으로 교체하려고 할수도 있다. 이것은 simplifyWhiteSpace()함수가 수행한다.

```
QString str = " BOB \t THE \nDOG \n";
cerr << str.simplifyWhiteSpace().ascii() << endl;
```

simplifyWhiteSpace()이 돌려준 문자렬은 다음과 같다.

B	O	B		T	H	E		D	O	G

QStringList::split()에 의하여 문자렬을 부분문자렬들로 분리할수 있다.

```
QString str = "polluter pays principle";
QStringList words = QStringList::split(" ", str);
```

우의 실례에서는 문자렬 "polluter pays principle"을 부분문자렬들인 "polluter", "pays", "principle"로 분리한다. split()함수는 빈 부분문자렬을 무시하는가(기정) 무시하지 않는가를 지

정하는 세번째의 bool형인수를 가지고있다.

join()에 의해 QStringList의 요소들을 결합하여 하나의 문자렬을 구성할수 있다. join()의 인수는 매개 쌍의 결합문자렬들사이에 삽입된다. 례를 들면 여기에 자모순으로 정렬되고 새행으로 구분되여 QStringList안에 포함되여있는 모든 문자렬들로 이루어지는 하나의 문자렬을 창조하는 방법이 있다.

```
words.sort();str = words.join("\n");
```

문자렬들을 취급할 때 흔히 문자렬이 비였는가 비지 않았는가를 결정해야 한다. 이것을 시험하는 한가지 수법은 isEmpty()를 호출하는것이며 다른 수법은 length()가 0인가 검사하는것이다.

QString은 null문자렬과 빈 문자렬을 구별한다. 이것은 0(null지적자)과 ""(빈문자렬)사이를 구별하는 C언어에 뿌리를 두고있다. 문자렬이 null인가 시험하기 위하여 isNull()을 호출할수 있다. 대부분의 응용프로그람들에서 문자렬이 어떤 문자들을 포함하는가 하는 문제가 제기된다. isEmpty()함수는 문자렬에 문자가 없으면(null이거나 비였으면) true를, 그렇지 않으면 false를 돌려줌으로써 이러한 정보를 제공한다.

const char * 문자렬과 QString사이의 변환은 대부분의 경우에 자동이다. 례를 들면

```
str += " (1870)";
```

여기서 형식화없이 const char *를 QString에 추가한다.

일부 상황에서는 const char *와 QString사이의 명시적변환이 필요하다. QString을 const char *로 변환하기 위하여서는 ascii()나 latin1()을 사용한다. 다른 방법으로 변환하려면 QString강제변환을 사용해야 한다.

QString에 대하여 ascii()나 latin1()을 호출하거나 const char *에로의 자동변환이 이 작업을 수행하게 할 때 돌려주는 문자렬은 QString객체가 소유한다. 이것은 기억루실에 대하여 걱정할 필요가 없다는것을 의미하며 Qt는 기억기를 반환한다. 다른 한편 지적자를 너무 오래 사용하지 말아야 한다. 례를 들면 원래의 QString을 수정한다면 지적자는 유효라고 담보할수 없다. 임의의 시간동안 const char *를 보관해야 한다면 그것을 QByteArray이나 QCString형의 변수에 대입할수 있다. 그러면 자료의 완전한 사본을 보관할수 있다.

QString은 암시적으로 공유된다. 이것은 QString의 복사가 단일지적자의 복사만큼 빠르다는것을 의미한다. 사본들중 하나가 변경되면 자료는 실제로 복사되고 이것은 배경에서 자동적으로 처리된다. 이러한 리유로 암시적공유는 흔히 《써넣기할 때 복사하는것》이라고 말한다.

암시적공유의 우점은 최적화이며 프로그람작성자의 개입을 요구하지 않고 간단히 수행된다.

Qt는 QBrush, QFont, QPen, QPixmap, QMap<K, T>, QValueList<T>, QValueVector<T>를 비롯한 다른 많은 클라스들에서도 암시적공유를 사용한다. 이것은 이러한 클라스들을 함수파라메터로서 그리고 돌림값으로서 값에 의해 넘기는것이 아주 효과적이게 한다.

C++는 강하게 분류되는 언어이며 이것은 형안전과 효과성을 비롯하여 많은 리득을 제공한다. 그러나 일부 상황에서는 일반적으로 자료를 보관할수 있어야 하며 이때 편리한 한가지 수법은 문자렬을 사용하는것이다. 례를 들면 문자렬은 본문값이나 수값을 문자렬형식으로 보관할수 있다. Qt는 서로 다른 형들을 보관하는 변수를 처리하는 훨씬 더 명백한 수단으로서 QVariant를 제공한다.

QVariant클라스는 QBrush, QColor, QCursor, QDateTime, QFont, QKeySequence, QPalette, QPen, QPixmap, QPoint, QRect, QRegion, QSize, QString을 비롯한 많은 Qt형의 값들을 보관할수 있다. QVariant클라스는 또한 용기들 즉 QMap<QString,QVariant>, QStringList, QValueList<QVariant>를 보관할수 있다. 4장의 표계산프로그람실현에서 QVariant를 사용하여 QString, double, 혹은 무효값일수 있는 세포의 값을 보관하였다.

가변형의 한가지 일반적인 사용은 건으로서 문자렬, 값으로서 가변형을 사용하는 매프이다. 환경구성자료는 보통 QSettings를 사용하여 보관하고 얻지만 일부 응용프로그람은 이 자료를 자료기지에 보관하여 직접 처리할수 있다. QMap<QString,QVariant>는 그러한 상황에서 리상적이다.

```
QMap<QString, QVariant> config;
config["Width"] = 890;
config["Height"] = 645;
config["ForegroundColor"] = black;
config["BackgroundColor"] = lightGray;
config["SavedDate"] = QDateTime::currentDateTime();
QStringList files;
files << "2003-05.dat" << "2003-06.dat" << "2003-07.dat";
config["RecentFiles"] = files;
```

암시적공유의 작업방법

암시적공유는 배경에서 자동적으로 작업하므로 암시적공유를 사용하는 클라스들을 사용할 때 자기 코드에서 이러한 최적화를 발생시키는 조작을 하지 말아야 한다. 그러나 동작을 아는것이 좋으므로 하나의 실례를 고찰하여 배경에서 무엇이 발생하는가를 보려고 한다.

```
QString str1 = "Humpty";
QString str2 = str1;
```

str1에 "Humpty"을 설정하고 str2을 str1과 같게 설정한다. 이 시점에서 두개의 QString이 기억기의 같은 자료구조(QStringData형)를 가리킨다. 문자자료에 대하여 자료구조는 같은 자료구조를 가리키는 QString이 몇개인가를 나타내는 참고계수를 보관한다. str1와 str2이 둘다 같은 자료를 가리키므로 참고계수는 2이다.

```
str2[0] = 'D';
```

str2을 수정할 때 우선 자료의 깊은 사본을 만들고 str1와 str2이 다른 자료구조를 가리키

도록 하여 자료가 자기 사본을 변경하도록 한다. str1자료("Humpty")의 참고계수는 1로 되고 str2자료("Dumpty")의 참고계수는 1로 설정된다. 참고계수1은 자료가 공유되지 않는다는것을 의미한다.

```
str2.truncate(4);
```

다시 str2을 수정하면 str2자료의 참고계수가 1로 되므로 복사가 생기지 않는다. truncate() 함수는 str2의 자료에 직접 조작하며 결과 문자렬 "Dump"이 생긴다. 참고계수는 여전히 1이다.

```
str1 = str2;
```

str2을 str1에 대입할 때 str1자료의 참고계수는 0으로 줄어든다. 이것은 "Humpty"자료를 사용하고있는 QString이 없다는것을 의미한다. 그다음 자료는 기억기로부터 해방된다. 두 QString은 현재 "Dump"를 가리키고 현재 참고계수는 2이다.

암시적으로 공유되는 클라스들을 쓰는것은 그리 어렵지 않다.

가변값들을 보관하는 매프를 순환하는것은 일부 값들이 용기이라면 아주 까다로울수 있다. type()를 사용하여 가변값을 보관하는 형을 검사함으로써 적당히 응답할수 있다.

```
QMap<QString, QVariant>::const_iterator it = config.begin();
while (it != config.end()) {
    QString str;
    if (it.data().type() == QVariant::StringList)
        str = it.data().toStringList().join(", ");
    else str = it.data().toString();
        cerr << it.key().ascii() << ": " << str.ascii() << endl;
    ++it;
}
```

용기형의 값들을 보관함으로써 QVariant을 사용하는 임의의 복합자료구조를 창조할수 있다.

```
QMap<QString, QVariant> price;
price["Orange"] = 2.10;
price["Pear"].asMap()["Standard"] = 1.95;
price["Pear"].asMap()["Organic"] = 2.25;
price["Pineapple"] = 3.85;
```

여기서는 문자렬건(제품명)과 류점수(가격) 혹은 매프로 된 값들을 가지는 매프를 창조하였다. 제일웃준위매프는 3개의 건 "Orange", "Pear", "Pineapple"을 포함한다. "Pear"건과 련상된 값은 두개 건("Standard"과 "Organic")을 포함하는 매프이다.

이러한 자료구조의 창조는 자기가 좋아하는 수법으로 자료구조를 만들수 있으므로 아주 매혹적일수 있다. 그러나 QVariant의 관례는 큰 비용을 요구한다. 읽기편리하게 보통 적당한

C++클라스를 정의하여 자료를 보관하는것이 좋다. 사용자정의클라스는 형안전을 제공하고 또한 QVariant보다 더 빠르고 기억기사용에서 효과적이다.

제12장. 자료기지

Qt의 SQL모듈은 SQL자료기지들을 호출하기 위한 가동환경 및 자료기지에 의존하지 않는 대면부와 자료기지들을 사용자대면부에 통합하기 위한 클라스들의 모임을 제공한다.

이 장에서는 우선 자료기지련결을 여는 방법과 자료기지에 대하여 임의의 SQL문을 실행하는 방법을 보여준다. 2절과 3절에서는 사용자대면부를 통하여 자료기지를 표시하고 수정하는 방법과 QDataTable에 의해 표창문부품의 자료를 표시하는 방법, QSqlForm에 의해 폼으로서 자료를 제시하는 방법을 사용자에게 제공하는데 초점을 둔다. 이 클라스들은 서로 훌륭히 교제하도록 설계되여있으며 주–세부보기와 구멍내기와 같은 일반적인 자료기지기능을 만들고 실현하기 쉽게 한다.

제1절. 련결과 질문

SQL질문을 실행하려면 우선 자료기지와의 련결을 확립해야 한다. 일반적으로 자료기지련결은 응용프로그람을 기동할 때 호출하는 개별적인 함수에서 설정된다. 례를 들면

```
bool createConnection()
{
    QSqlDatabase *db = QSqlDatabase::addDatabase("QOCI8");
    db->setHostName("mozart.konkordia.edu");
    db->setDatabaseName("musicdb");
    db->setUserName("gbatstone");
    db->setPassword("T17aV44");
    if (!db->open()) {
        db->lastError().showMessage();
        return false;
    }
    return true;
}
```

우선 QSqlDatabase::addDatabase()를 호출하여 QSqlDatabase객체를 창조한다. addDatabase()의 인수는 Qt가 자료기지호출에 사용해야 할 자료기지구동프로그람을 지정한다. 이 경우에는 Oracle을 사용한다. 상업판의 Qt 3.2에는 다음과 같은 구동프로그람들이 포함되여있다. 즉 QODBC3(ODBC), QOCI8(Oracle), QTDS7(Sybase Adaptive Server), QPSQL7(PostgreSQL), QMYSQL3(MySQL), 그리고 QDB2(IBM DB2). 무료 및 비상업판들은 이것들의 부분모임을 포함한다.

다음으로 자료기지주콤퓨터이름, 자료기지이름, 사용자이름, 그리고 암호를 설정하고 련결을 열려고 시도한다. open()이 실패하면 QSqlError::showMessage()을 리용하여 오유통보문을 표

시한다.

일반적으로 main()에서 createConnection()를 호출할수 있다.

```
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    if (!createConnection())
        return 1;
    ...
    return app.exec();
}
```

일단 련결이 확립되면 QSqlQuery을 사용하여 기초하고있는 자료기지가 유지하는 SQL문을 실행할수 있다. 례를 들면 여기에 SELECT문의 실행방법이 있다.

```
QSqlQuery query;
query.exec("SELECT title, year FROM cd WHERE year >= 1998");
```

exec()호출후에 질문의 결과모임을 항행할수 있다.

```
while (query.next()) {
    QString title = query.value(0).toString();
    int year = query.value(1).toInt();
    cerr << title.ascii() << ": " << year << endl;
}
```

next()를 한번 호출하여 QSqlQuery가 결과모임의 첫 레코드에 위치하게 한다. next()의 련이은 호출은 끝에 이를 때까지 매번 한 레코드씩 레코드지적자를 전진시킨다. 끝점에서 next()는 false를 돌려준다. 결과모임이 비였으면 next()의 첫 호출은 false를 돌려준다.

value()함수는 마당값을 QVariant로서 돌려준다. 마당들에는 SELECT문에서 주어지는 순서로 0으로부터 번호가 붙는다. QVariant클라스는 int와 QString을 비롯한 수많은 C++와 Qt형들을 제공한다. 자료기지에 보관할수 있는 각이한 자료형들은 대응하는 C++ 및 Qt형들로 변환되고 QVariant들에 보관된다. 례를 들면 VARCHAR는 QString으로, DATETIME은 QDateTime으로 표시된다.

QSqlQuery는 결과모임을 항행하기 위한 다른 함수 즉 first(), last(), prev(), seek(), at()를 제공한다. 이러한 함수들은 편리하지만 일부 자료기지들에서 기억기가 모자란다. 큰 자료모임에 조작할 때 간단한 최적화를 위하여 exec()를 호출하기전에 QSqlQuery::setForwardOnly(true)를 호출할수 있으며 오직 결과모임을 항행할 때만 next()를 사용한다.

초기에 exec()의 인수로서 SQL질문을 지정하였으나 곧 실행하는 구성자에 SQL질문을 직접 넘길수 있다.

```
QSqlQuery query("SELECT title, year FROM cd WHERE year >= 1998");
```

여기에 오유를 검사하고 문제가 발생하는 경우에 QMessageBox를 펼치는 방법이 있다.

```
if (!query.isActive())
    query.lastError().showMessage();
```

INSERT는 SELECT처럼 대체로 실행하기 쉽다.

```
QSqlQuery query("INSERT INTO cd (id, artistid, title, year) "
    "VALUES (203, 102, 'Living in America', 2002)");
```

그다음에 QSqlQuery::numRowsAffected()는 SQL문의 영향을 받은 행수(혹은 자료기지가 그 정보를 제공할수 없으면 –1)를 돌려준다..

많은 레코드를 삽입해야 하거나 값들을 문자렬로 변환하지 않으려면(그리고 값들을 정확히 확장하려면) prepare()를 사용하여 대리기호를 포함하는 질문을 지정한 다음 삽입하려는 값들을 속박할수 있다. Qt는 모든 자료기지에서 Oracle형식과 ODBC형식의 대리기호문법을 유지한다. 이때 그 문법이 유효하면 그대로 리용하고 그렇지 않으면 모의한다. 여기에 이름있는 대리기호를 가지는 Oracle형식문법을 사용하는 실례가 있다.

```
QSqlQuery query(db);
query.prepare("INSERT INTO cd (id, artistid, title, year) "
                                "VALUES (:id, :artistid, :title, :year)");
query.bindValue(":id", 203);
query.bindValue(":artistid", 102);
query.bindValue(":title", QString("Living in America"));
query.bindValue(":year", 2002);
query.exec();
```

여기에 ODBC형식의 위치대리기호를 리용하는 실례가 있다.

```
QSqlQuery query(db);
query.prepare("INSERT INTO cd (id, artistid, title, year) VALUES (?, ?, ?, ?)");
query.addBindValue(203);
query.addBindValue(102);
query.addBindValue(QString("Living in America"));
query.addBindValue(2002);
query.exec();
```

prepare()호출후에 bindValue()나 addBindValue()를 호출하여 새 값들을 속박한 다음 다시 exec()를 호출하여 새 값들로 질문을 실행할수 있다.

대리기호는 흔히 2진자료 혹은 비ASCII 혹은 비Latin-1문자들을 포함하는 문자렬을 지정하는데 쓰인다. 이러한 배경하에서 Qt는 유니코드를 유지하는 자료기지들에서 유니코드를 사용하고 그렇지 않은것들에 대해서는 문자렬을 적당한 부호화로 명백히 변환한다.

Qt는 사용가능한 자료기지들에 대하여 SQL일괄처리를 유지한다. 일괄처리를 시작하려면

자료기지련결을 표시하는 QSqlDatabase객체에 대하여 transaction()을 호출한다. 일괄처리를 완료하려면 commit() 혹은 rollback()를 호출한다. 례를 들면 여기에 일괄처리내에서 외부열쇠를 찾고 INSERT문을 실행하는 방법이 있다.

```
QSqlDatabase::database()->transaction();
QSqlQuery query;
query.exec("SELECT id FROM artist WHERE name = 'Gluecifer'");
if (query.next()) {
    int artistId = query.value(0).toInt();
    query.exec("INSERT INTO cd (id, artistid, title, year) "
               "VALUES (201, " + QString::number(artistId)+ ", 'Riding the Tiger', 1997)");
}
QSqlDatabase::database()->commit();
```

QSqlDatabase::database()함수는 createConnection()에서 생성한 QSqlDatabase객체의 지적자를 돌려준다. 일괄처리를 기동할수 없으면 QSqlDatabase::transaction()은 false를 돌려준다.

일부 자료기지들은 일괄처리를 유지하지 않는다. 그러한 경우에 transaction(), commit(), rollback()함수들은 아무것도 수행하지 않는다. 자료기지와 련결된 QSqlDriver에 대하여 hasFeature()를 리용하여 자료기지가 일괄처리를 유지하는가 시험할수 있다.

```
QSqlDriver *driver = QSqlDatabase::database()->driver();
if (driver->hasFeature(QSqlDriver::Transactions)) ...
```

지금까지의 실례들에서는 응용프로그람이 단일한 자료기지련결을 리용하고있다고 가정하였다. 다중련결을 사용하려고 한다면 addDatabase()에 둘째인수로서 이름을 넘길수 있다. 례를 들면

```
QSqlDatabase *db = QSqlDatabase::addDatabase("QPSQL7", "OTHER");
db->setHostName("saturn.mcmanamy.edu");
db->setDatabaseName("starsdb");
db->setUserName("gilbert");
db->setPassword("ixtapa6");
```

그다음 이름을 넘기여 QSqlDatabase객체의 지적자를 얻을수 있다.

```
QSqlDatabase::database():
QSqlDatabase *db = QSqlDatabase::database("OTHER");
```

다른 련결을 리용하여 질문을 실행하려면 QSqlDatabase객체를 QSqlQuery구성자에 넘겨야 한다.

```
QSqlQuery query(db);
query.exec("SELECT id FROM artist WHERE name = 'Mando Diao'");
```

매개의 련결이 오직 하나의 능동일괄처리만 처리할수 있으므로 다중련결은 한번에 하나

이상의 일괄처리를 수행하려고 할 때 사용할수 있다. 다중자료기지련결을 사용한다면 여전히 하나의 이름없는 련결을 가질수 있으며 QSqlQuery은 아무것도 지정되지 않으면 그 련결을 사용한다.

QSqlQuery외에도 Qt는 고급한 클라스로서 QSqlCursor클라스를 제공한다. 이 클라스는 QSqlQuery를 계승하고 편의함수들을 확장하여 가장 일반적인 SQL조작 SELECT, INSERT, UPDATE, DELETE를 수행하는 본래의 SQL을 입력하는것을 피할수 있다. 또한 QSqlCursor는 QDataTable을 자료기지에 속박하는 클라스이다. 여기서는 QSqlCursor를 설명하고 다음 절에서는 자료기지를 인식하는 QTable의 파생클라스 QDataTable을 설명한다.

여기에 QSqlCursor을 리용하여 SELECT를 처리하는 실례가 있다.

```
QSqlCursor cursor("cd");
cursor.select("year >= 1998");
```

등가한 QSqlQuery은 다음과 같다.

```
QSqlQuery query("SELECT id, artistid, title, year FROM cd " "WHERE year >= 1998");
```

결과모임의 항행은 QSqlQuery에서와 같은데 마당번호대신에 마당이름들을 value()에 넘길수 있다.

```
while (cursor.next()) {
      QString title = cursor.value("title").toString();
      int year = cursor.value("year").toInt();
      cerr << title.ascii() << ": " << year << endl;
}
```

표에 레코드를 삽입하기 위하여 우선 newQSqlRecord의 지적자를 돌려주는 primeInsert()를 호출하여야 한다. 그다음 설정하려고 하는 QSqlRecord안의 매개 마당에 대하여 setValue()를 호출하고 insert()를 호출하여 QSqlRecord의 자료를 자료기지에 삽입한다. 례를 들면

```
QSqlCursor cursor("cd");
QSqlRecord *buffer = cursor.primeInsert();
buffer->setValue("id", 113);
buffer->setValue("artistid", 224);
buffer->setValue("title", "Shanghai My Heart");
buffer->setValue("year", 2003);
cursor.insert();
```

레코드를 갱신하려면 우선 수정하려는 레코드에 QSqlCursor를 배치해야 한다. (례를 들면 select()와 next()를 리용한다.) 그다음 primeUpdate()를 리용하여 레코드자료의 사본을 포함하는 QSqlRecord의 지적자를 얻는다. 그다음 setValue()를 리용하여 변경하려는 마당들을 설정하고 update()를 호출하여 이 변경을 자료기지에 써넣는다. 례를 들면

```
QSqlCursor cursor("cd");
```

```
cursor.select("id = 125");
if (cursor.next()) {
    QSqlRecord *buffer = cursor.primeUpdate();
    buffer->setValue("title", "Melody A.M.");
    buffer->setValue("year", buffer->value("year").toInt() + 1);
    cursor.update();
}
```

레코드의 삭제는 갱신과 비슷하지만 더 간단하다.

```
QSqlCursor cursor("cd");
cursor.select("id = 128");
if (cursor.next()) {
    cursor.primeDelete();
    cursor.del();
}
```

QSqlQuery과 QSqlCursor클라스들은 Qt와 SQL자료기지사이의 대면부를 제공한다. 다음의 2개 절에서는 GUI응용프로그람안에서 그 클라스들을 사용하여 사용자가 자료기지에 보관된 자료를 표시하고 교제하는 방법을 알게 된다.

제2절. 표형식의 폼에서 자료의 표시

QDataTable클라스는 열람과 편집기능을 가지는 자료기지인식QTable창문부품이다. 이 클라스는 QSqlCursor를 통하여 자료기지와 교제한다. 여기서는 QDataTable을 사용하는 2개의 대화칸을 고찰한다. 다음 절에서 제시하는 QSqlForm에 기초하는 대화칸와 함께 이 폼들은 CD Collection응용프로그람을 구성한다.

응용프로그람은 다음과 같이 정의된 3개의 표를 사용한다.

```
CREATE TABLE artist ( id INTEGER PRIMARY KEY,
    name VARCHAR(40) NOT NULL, country VARCHAR(40));
CREATE TABLE cd ( id INTEGER PRIMARY KEY,
    artistid INTEGER NOT NULL, title VARCHAR(40) NOT NULL,
    year INTEGER NOT NULL, FOREIGN KEY (artistid) REFERENCES artist);
CREATE TABLE track ( id INTEGER PRIMARY KEY,
    cdid INTEGER NOT NULL, number INTEGER NOT NULL,
    title VARCHAR(40) NOT NULL, duration INTEGER NOT NULL,
    FOREIGN KEY (cdid) REFERENCES cd);
```

일부 자료기지는 외부열쇠를 제공하지 않는다. 이러한 자료기지에서는 FOREIGN KEY부들을 삭제해야 한다. 여전히 실례는 작업하지만 자료기지는 참고완전성을 가지지 않는다.

artist
id
name
country

1:N

cd
id
artistid
title
year

1:N

track
id
cdid
number
title
duration

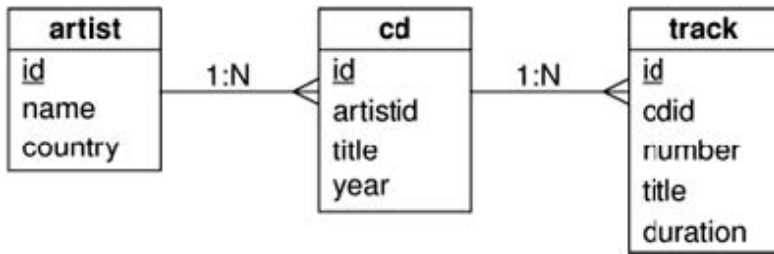



그림 12-1. CD Collection응용프로그람의 표들

처음에 작성하는 클라스는 사용자가 기사목록을 편집하게 하는 대화칸이다. 사용자는 QDataTable의 상황차림표를 리용하여 기사들을 삽입, 갱신, 혹은 삭제할수 있다. 이러한 변경은 사용자들이 Update를 찰칵할 때 자료기지에 적용된다.

Update Artists

	Name	Country
1	Gluecifer	Norway
2	The Sounds	Sweden
3	Melvins	
4	Strapping Young Lad	

Insert
Update
Delete

Update Cancel

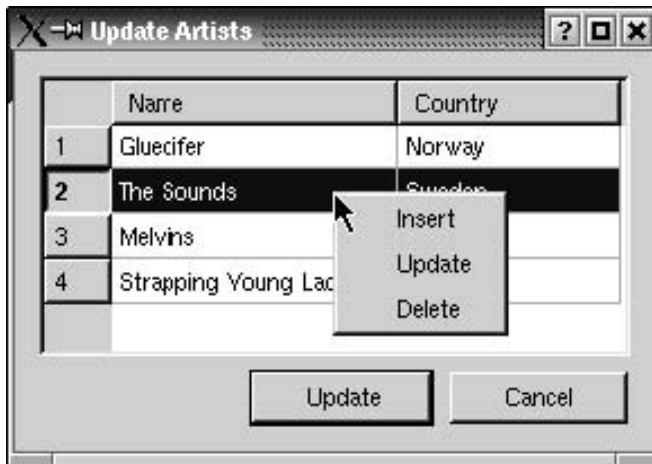



그림 12-2. ArtistForm대화칸

여기에 대화칸의 클라스정의가 있다.

```
class ArtistForm : public QDialog
{
    Q_OBJECT
public:
    ArtistForm(QWidget *parent = 0, const char *name = 0);
protected slots:
    void accept();
    void reject();
private slots:
    void primeInsertArtist(QSqlRecord *buffer);
    void beforeInsertArtist(QSqlRecord *buffer);
    void beforeDeleteArtist(QSqlRecord *buffer);
private:
```

```
    QSqlDatabase *db;
    QDataTable *artistTable;
    QPushButton *updateButton;
    QPushButton *cancelButton;
};
```

accept()와 reject()처리부들은 QDialog로부터 재정의된다.

```
ArtistForm::ArtistForm(QWidget *parent, const char *name) : QDialog(parent, name)
{
    setCaption(tr("Update Artists"));
    db = QSqlDatabase::database("ARTIST");
    db->transaction();
    QSqlCursor *artistCursor = new QSqlCursor("artist", true, db);
    artistTable = new QDataTable(artistCursor, false, this);
    artistTable->addColumn("name", tr("Name"));
    artistTable->addColumn("country", tr("Country"));
    artistTable->setAutoDelete(true);
    artistTable->setConfirmDelete(true);
    artistTable->setSorting(true);
    artistTable->refresh();
    updateButton = new QPushButton(tr("Update"), this);
    updateButton->setDefault(true);
    cancelButton = new QPushButton(tr("Cancel"), this);
```

ArtistForm구성자에서는 ARTIST자료기지련결을 리용하여 일괄처리를 시작한다. 그다음 자료기지의 기사표에 QSqlCursor를 창조하고 그것을 현시할 QDataTable를 창조한다.

QSqlCursor구성자의 둘째 인수는《자동거주》기발이다. true를 넘기여 QSqlCursor에 표의 매개 마당에 대한 정보를 적재하고 모든 마당들에 조작하게 한다.

QDataTable구성자의 둘째인수도 역시 자동거주기발이다. true이면 QDataTable는 자동적으로 QSqlCursor의 결과모임안의 매개 마당용의 렬들을 창조한다. false를 넘기고 addColumn()를 호출하여 결과모임의 name과 country마당들에 대응하는 2개 렬을 제공한다.

setAutoDelete()를 호출하여 QSqlCursor의 소유자를 QDataTable에 넘기므로 그것을 자체로 삭제할 필요가 없다. setConfirmDelete()를 호출하여 QDataTable이 사용자에게 삭제를 확인하게 하는 통보창을 펼친다. setSorting(true)를 호출하여 사용자가 렬제목우에서 찰칵하여 렬에 따라서 표를 정렬하게 한다. 끝으로 refresh()를 호출하여 QDataTable에 자료기지의 자료를 채운다.

또한 Update와 Cancel단추를 창조한다.

```
    connect(artistTable, SIGNAL(beforeDelete(QSqlRecord *)), this,
```

```
            SLOT(beforeDeleteArtist(QSqlRecord *)));
        connect(artistTable, SIGNAL(primeInsert(QSqlRecord *)), this,
            SLOT(primeInsertArtist(QSqlRecord *)));
        connect(artistTable, SIGNAL(beforeInsert(QSqlRecord *)), this,
            SLOT(beforeInsertArtist(QSqlRecord *)));
        connect(updateButton, SIGNAL(clicked()), this, SLOT(accept()));
        connect(cancelButton, SIGNAL(clicked()), this, SLOT(reject()));
```

QDataTable의 3개 신호를 3개의 비공개처리부들에 련결한다. Update단추를 accept()에, Cancel단추를 reject()에 련결한다.

```
        QHBoxLayout *buttonLayout = new QHBoxLayout;
        buttonLayout->addStretch(1);
        buttonLayout->addWidget(updateButton);
        buttonLayout->addWidget(cancelButton);
        QVBoxLayout *mainLayout = new QVBoxLayout(this);
        mainLayout->setMargin(11);
        mainLayout->setSpacing(6);
        mainLayout->addWidget(artistTable);
        mainLayout->addLayout(buttonLayout);
    }
```

끝으로 QPushButton들을 수평배치관리자에 넣고 QDataTable와 수평배치관리자를 수직배치관리자에 넣는다.

```
    void ArtistForm::accept()
    {
        db->commit();
        QDialog::accept();
    }
```

사용자가 Update를 찰칵하면 일괄처리를 예약하고 기초클라스의 accept()함수를 호출한다.

```
    void ArtistForm::reject()
    {
        db->rollback();
        QDialog::reject();
    }
```

사용자가 Cancel을 찰칵하면 일괄처리를 되돌리고 기초클라스의 reject()함수를 호출한다.

```
    void ArtistForm::beforeDeleteArtist(QSqlRecord *buffer)
    {
```

```
        QSqlQuery query(db);
        query.exec("DELETE FROM track WHERE track.id IN (SELECT track.id FROM track, "
            " cd WHERE track.cdid = cd.id AND cd.artistid = "
            + buffer->value("id").toString() + ")");
        query.exec("DELETE FROM cd WHERE artistid = " + buffer->value("id").toString());
    }
```

beforeDeleteArtist()처리부는 레코드가 삭제되기 직전에 발생되는 QDataTable의 beforeDelete()신호에 련결된다. 여기서는 기사에 의하여 CD들로부터 모든 자리길들을 삭제할데 대한 질문과 기사에 의해 모든 CD들을 삭제할데 대한 질문을 실행함으로써 폭포식(종속적인) 삭제를 수행한다. 이러한 삭제는 관계완전성에 위험을 주지 않는다. 그것은 이러한 삭제가 폼의 구성자에서 시작되는 일괄처리의 상황에서 모두 수행되기때문이다.

또 다른 수법은 사용자가 cd표에 의해 참고되는 기사들을 삭제하지 않도록 하는것이다. 이것을 달성하려면 QDataTable::contextMenuEvent()를 재정의하여 삭제를 자체로 처리해야 한다. 자료기지가 관계완전성을 실시하도록 설정된 경우에 작업하는 초기의 수법은 단순히 삭제를 시도하고 자료기지에 그것을 맡기여 미연에 방지하는것이다.

```
    void ArtistForm::primeInsertArtist(QSqlRecord *buffer)
    {
        buffer->setValue("country", "DPRK");
    }
```

primeInsertArtist()처리부는 사용자가 새 레코드의 편집을 시작하기 직전에 발생되는 QDataTable의primeInsert()신호에 련결된다. 이 처리부를 리용하여 새 레코드의 country마당의 기정값을 우리 나라에서 사용하는 응용프로그람의 리상적인 기정값인 "DPRK"로 설정한다.

이것은 마당에 기정값을 설정하는 한가지 수법이다. 또 하나의 수법은 QSqlCursor의 파생클라스를 만들고 primeInsert()를 재정의하는것이다. 이 함수는 같은 응용프로그람에서 같은 QSqlCursor를 여러번 사용하고 일관한 동작을 담보하려고 하는 경우에 의의를 가진다. 셋째 수법은 CREATE TABLE문의 DEFAULT절을 사용하는 자료기지준위에서 수행하는것이다.

```
    void ArtistForm::beforeInsertArtist(QSqlRecord *buffer)
    {
        buffer->setValue("id", generateId("artist", db));
    }
```

beforeInsertArtist()처리부는 사용자가 새 레코드의 편집을 완료하고 그것을 보관하기 위하여 *Enter*를 눌렀을 때 발생되는 QDataTable의 beforeInsert()신호에 련결된다. id마당의 값을 생성된 값으로 설정한다. generateId()라는 함수에 기초하여 유일한 주열쇠(primary key)를 생성한다.

generateId()를 여러번 생성해야 하므로 머리부파일에서 이 함수를 inline으로 정의하고 그

것이 요구될 때마다 포함한다. 여기에 그것을 실현하는 빠른(비효과적인) 수법이 있다.

```
inline int generateId(const QString &table, QSqlDatabase *db)
{
    QSqlQuery query(db);
    query.exec("SELECT max(id) FROM " + table);
    query.next();
    return query.value(0).toInt() + 1;
}
```

generateId()함수는 대응하는 INSERT문과 같은 일괄처리의 상황에서 실행된다면 정확한 동작을 담보할수 있다.

일부 자료기지는 자동생성되는 마당들을 유지한다. 이 마당들과 관련하여 단지 자료기지에 id마당들을 자동생성한다고 알리고 QSqlCursor에 대하여 setGenerated("id", false)를 호출하여 id마당의 값을 생성하지 않는다고 말한다.

이제 QDataTable를 사용하는 다른 대화칸을 고찰한다. 이 대화칸에서는 주-세부보기를 실현한다. 주(기본)보기는 CD들의 목록이다. 세부보기는 현재 CD의 자리길목록이다. 이 대화칸은 CD Collection응용프로그람의 기본창문이다.

이번에는 Add, Edit, Delete단추들을 제공하여 사용자가 상황차림표에 기초하지 않고 CD목록을 수정하게 한다. 사용자들이 *Add* 혹은 *Edit*를 찰칵하면 CdForm대화칸을 펼친다. (CdForm은 다음 절에서 설명한다.)

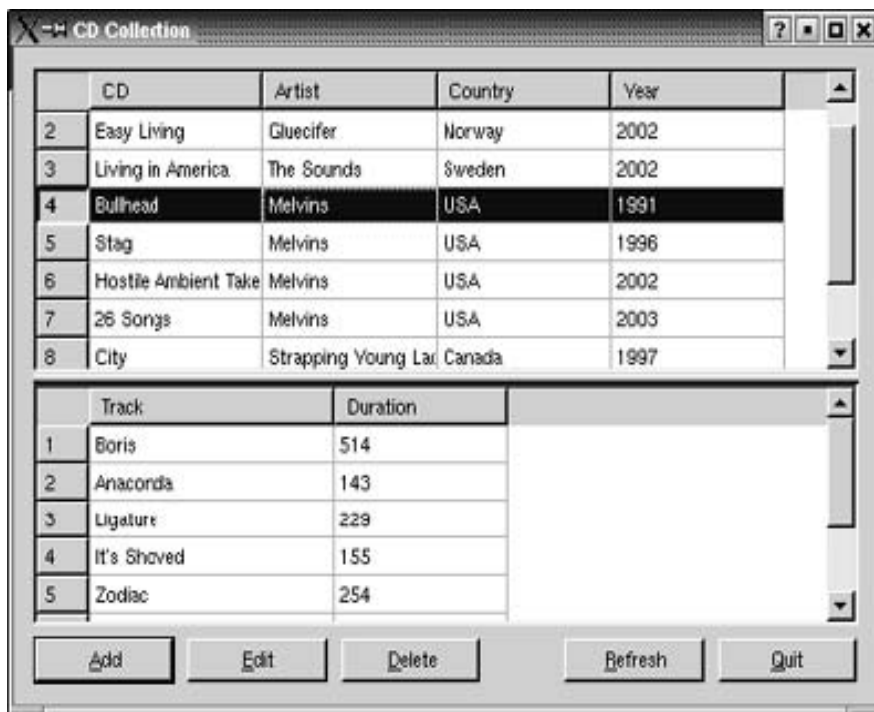



그림 12-3. MainForm대화칸

이 실례와 이전 실례사이의 또 하나의 차이는 외부열쇠를 해결하여 기사의 ID가 아니라 기사의 이름과 나라를 표시할수 있다는것이다. 이것을 달성하기 위하여 임의의 SELECT문(이 경우에는 결합)들을 유지하는 QSqlCursor의 파생클라스인 QSqlSelectCursor를 사용해야 한다.

우선 클라스정의는 다음과 같다.

```
class MainForm : public QDialog
{
    Q_OBJECT
public:
    MainForm(QWidget *parent = 0, const char *name = 0);
private slots:
    void addCd();
    void editCd();
    void deleteCd();
    void currentCdChanged(QSqlRecord *record);
private:
    QSplitter *splitter;
    QDataTable *cdTable;
    QDataTable *trackTable;
    QPushButton *addButton;
    ...
    QPushButton *quitButton;
};
```

MainForm클라스는 QDialog을 계승한다.

```
MainForm::MainForm(QWidget *parent, const char *name) : QDialog(parent, name)
{
    setCaption(tr("CD Collection"));
    splitter = new QSplitter(Vertical, this);
    QSqlSelectCursor *cdCursor = new QSqlSelectCursor("SELECT cd.id, title, name, "
        "country, year FROM cd, artist WHERE cd.artistid = artist.id");
    if (!cdCursor->isActive()) {
      QMessageBox::critical(this, tr("CD Collection"),tr("The database has not been created.\n"
          "Run the cdtables example to create a sample
          "database, then copy cdcollection.dat into "
          "this directory and restart this application."));
```

```
        qApp->quit();
    }
    cdTable = new QDataTable(cdCursor, false, splitter);
    cdTable->addColumn("title", tr("CD"));
    cdTable->addColumn("name", tr("Artist"));
    cdTable->addColumn("country", tr("Country"));
    cdTable->addColumn("year", tr("Year"));
    cdTable->setAutoDelete(true);
    cdTable->refresh();
```

구성자에서는 cd표의 읽기전용QDataTable과 그와 련관된 유표를 창조한다. 유표는 cd표와 기사표들을 결합하는 질문에 기초하고있다. QDataTable은 QSqlSelectCursor에 조작해야 하므로 읽기전용이다. 읽기전용표들은 상황차림표를 제공한다.

유표질문이 실패하면 통보창을 펼치고 오유가 발생하였으며 응용프로그람을 완료한다는 것을 표시한다.

```
    QSqlCursor *trackCursor = new QSqlCursor("track");
    trackCursor->setMode(QSqlCursor::ReadOnly);
    trackTable = new QDataTable(trackCursor, false, splitter);
    trackTable->setSort(trackCursor->index("number"));
    trackTable->addColumn("title", tr("Track"));
    trackTable->addColumn("duration", tr("Duration"));
```

둘째QDataTable과 그 유표를 창조한다. 유표에 대하여 setMode(QSqlCursor::ReadOnly)를 호출함으로써 표를 읽기전용으로 만들고 setSort()를 호출하여 자리길들을 자리길번호에 따라 정렬한다.

```
    addButton = new QPushButton(tr("&Add"), this);
    editButton = new QPushButton(tr("&Edit"), this);
    deleteButton = new QPushButton(tr("&Delete"), this);
    refreshButton = new QPushButton(tr("&Refresh"), this);
    quitButton = new QPushButton(tr("&Quit"), this);
    connect(addButton, SIGNAL(clicked()),this, SLOT(addCd()));
    ...
    connect(quitButton, SIGNAL(clicked()), this, SLOT(close()));
    connect(cdTable, SIGNAL(currentChanged(QSqlRecord *)),this,
        SLOT(currentCdChanged(QSqlRecord *)));
    connect(cdTable,SIGNAL(doubleClicked(int, int, int, const QPoint &)), this,
        SLOT(editCd()));
```

```
    ...
}
```

사용자대면부의 나머지부분을 설정하고 필요한 동작을 생성하는데 요구되는 신호–처리부 련결들을 창조한다.

```
void MainForm::addCd()
{
    CdForm form(this);
    if (form.exec()) {
        cdTable->refresh();
        trackTable->refresh();
    }
}
```

사용자가 Add를 찰칵할 때 이행금지CdForm대화칸을 펼치고 사용자가 그우에서 *Update*를 찰칵하면 QDataTable들을 갱신한다.

```
void MainForm::editCd()
{
    QSqlRecord *record = cdTable->currentRecord();
    if (record) {
        CdForm form(record->value("id").toInt(), this);
        if (form.exec()) {
            cdTable->refresh();
            trackTable->refresh();
        }
    }
}
```

사용자가 Edit를 찰칵하면 CdForm구성자에 현재CD의 ID를 인수로서 넘기여 이행금지 CdForm대화칸을 기동한다. 그러면 대화칸이 펼쳐지고 현재CD의 자료들이 들어있는 마당들이 표시된다.

폼을 여기서 수행한 ID로 파라메터화하면 ID는 폼이 나타날 때 유효로 되지 않는다. 례를 들면 어떤 사용자가 CD를 삭제하기전에 다른 사용자가 Edit를 잠간 찰칵할수 있다. CdForm에서 수행할수 있는것은 transaction()호출후에 즉시 넘기는 ID에 대하여 SELECT를 실행하고 여전히 ID가 존재한다면 전진하는것이다. 여기서는 단순히 자료기지에 기초하여 무효한 ID를 사용하려는 시도가 있으면 오유를 알리는것이다.

```
void MainForm::deleteCd()
{
```

```
        QSqlRecord *record = cdTable->currentRecord();
        if (record) {
            QSqlQuery query;
            query.exec("DELETE FROM track WHERE cdid = " + record->value("id").toString());
            query.exec("DELETE FROM cd WHERE id = " + record->value("id").toString());
            cdTable->refresh();
            trackTable->refresh();
        }
    }
```

사용자들이 Delete를 찰칵할 때 자리길표에서 현재CD의 모든 자리길들을 삭제한 다음 cd표에서 현재CD를 삭제한다. 그다음 두 표를 갱신한다.

```
    void MainForm::currentCdChanged(QSqlRecord *record)
    {
        trackTable->setFilter("cdid = " + record->value("id").toString());
        trackTable->refresh();
    }
```

currentCdChanged()처리부는 cdTable의 currentChanged()신호에 련결된다. 이 신호는 사용자가 현재CD를 수정하거나 사용자가 다른 CD를 현재CD로 만들려고 할 때 발생된다. 현재CD가 달라질 때마다 자리길표에 대하여 setFilter()를 호출하고 현재CD와 관련된 자리길들을 현시하도록 초기화하고 refresh()를 호출하여 표에 관련한 자료를 다시 옮기게 한다.

이것은 MainForm을 실현하는데 요구되는 코드의 전부이다. 한가지 가능한것은 매개 자리길의 지속시간을 초("155")가 아니라 분과 초(례를 들면 "02:35")로 나누어 표시할수 있는것이다. 그러기 위하여 QSqlCursor의 파생클라스를 만들고 calculateField()함수를 재정의하여 지속시간마당을 요구되는 형식의 QString으로 변환할수 있다.

```
    QVariant TrackSqlCursor::calculateField(const QString &name)
    {
        if (name == "duration") {
           int duration = value("duration").toInt();
           return QString("%1:%2").arg(duration / 60, 2).arg(duration % 60, 2);
        }
        return QVariant();
    }
```

또한 유표에 대하여 setCalculated("duration", true)을 호출하여 QDataTable에게 단순히 value()을 사용하지 않고 지속시간마당에 대하여 calculateField()가 돌려주는 값을 사용한다는것을 알린다.

제3절. 자료인식폼의 창조

Qt는 자료기지와 폼들사이의 교제를 혁신하는 수법을 제공한다. 각 기본창문부품에 대하여 제각기 자료기지허용판을 만드는것이 아니라 Qt는 자료기지마당들을 창문부품들과 련결하는데 QSqlForm와 QSqlPropertyMap를 사용하여 임의의 창문부품이 자료를 인식하게 만들수 있다. 기본창문부품이나 사용자정의창문부품은 이러한 클라스들을 사용하여 자료를 인식하게 만들수 있다.

QSqlForm은 폼들을 창조하여 자료기지안의 개별적인 레코드들을 간단히 열람하거나 편집하게 하는 QObject의 파생클라스이다. 일반적인 사용법은 다음과 같다.

① 레코드의 마당들에 대응하는 편집기창문부품들(QLineEdits, QComboBoxes, QSpinBoxes, 등)을 창조한다.

② QSqlCursor을 창조하고 그것을 편집하려는 레코드로 옮긴다.

③ QSqlForm객체를 창조한다.

④ QSqlForm에게 어느 편집기창문부품이 어느 자료기지마당에 결합되는가를 알린다.

⑤ QSqlForm::readFields()함수를 호출하여 현재레코드로부터 편집기창문부품들에 자료를 옮긴다.

⑥ 대화칸을 표시한다.

⑦ QSqlForm::writeFields()함수를 호출하여 갱신된 값을 자료기지에 복사한다.

이것을 설명하기 위하여 CdForm대화칸의 코드를 고찰한다. 이 대화칸은 사용자가 CD레코드를 창조하거나 편집하게 한다. 사용자는 CD의 제목, 기사, 제작년도와 매개 자리길의 제목과 연주시간을 지정할수 있다.

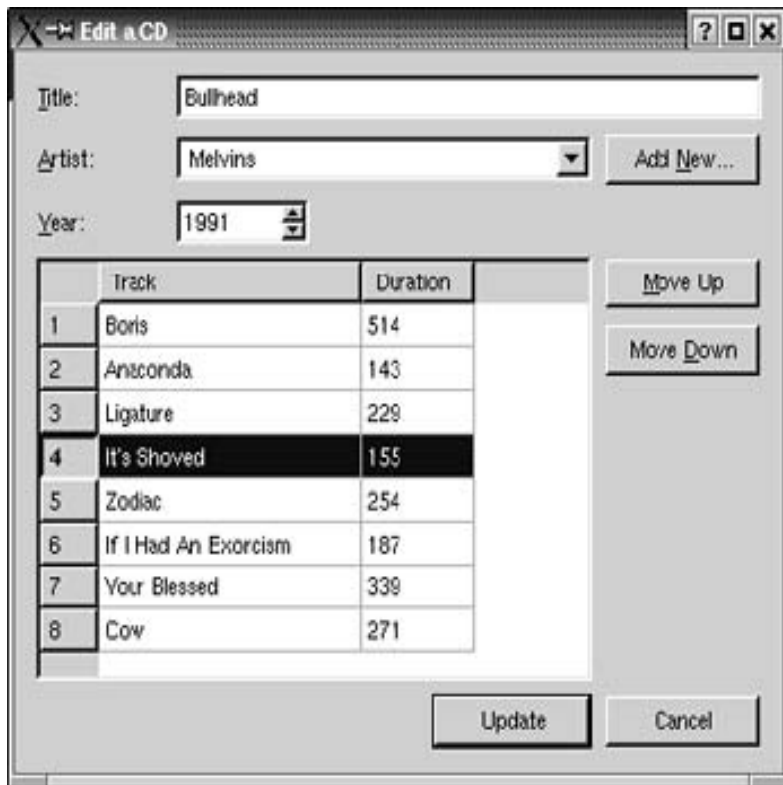



그림 12-4. CdForm대화칸

클라스정의부터 고찰하자.

```
class CdForm : public QDialog
{
    Q_OBJECT
public:
    CdForm(QWidget *parent = 0, const char *name = 0);
    CdForm(int id, QWidget *parent = 0, const char *name = 0);
    ~CdForm();
protected slots:
    void accept();
    void reject();
private slots:
    void addNewArtist();
    void moveTrackUp();
    void moveTrackDown();
    void beforeInsertTrack(QSqlRecord *buffer);
    void beforeDeleteTrack(QSqlRecord *buffer);
```

```
private:
      void init();
      void createNewRecord();
      void swapTracks(int trackA, int trackB);
      QLabel *titleLabel;
      QLabel *artistLabel;
      ...
      QDataTable *trackTable;
      QSqlForm *sqlForm;
      QSqlCursor *cdCursor;
      QSqlCursor *trackCursor;
      int cdId;
      bool newCd;
};
```

2개의 구성자를 선언한다. 즉 하나는 새 CD를 자료기지에 삽입하기 위한 구성자이고 다른하나는 현존CD를 갱신하기 위한 구성자이다. accept()와 reject()처리부들은 QDialog로부터 재정의된다.

```
CdForm::CdForm(QWidget *parent, const char *name) : QDialog(parent, name)
{
      setCaption(tr("Add a CD"));
      cdId = -1;init();
}
```

첫 구성자는 대화칸의 제목을 "Add a CD"로 설정하고 비공개init()함수를 호출하여 남은 일을 수행한다.

```
CdForm::CdForm(int id, QWidget *parent, const char *name) : QDialog(parent, name)
{
      setCaption(tr("Edit a CD"));
      cdId = id;
      init();
}
```

둘째 구성자는 제목을 "Edit a CD"로 설정하고 역시 init()를 호출한다.

```
void CdForm::init()
{
      db = QSqlDatabase::database("CD");
      db->transaction();
```

```
if (cdId == -1)
    createNewRecord();
```

init()에서는 CD자료기지련결을 리용하여 일괄처리를 시작한다. CdForm과 ArtistForm에서는 다른 련결들을 사용해야 한다. 그것은 두개 폼을 동시에 열고 한 폼이 다른 폼에 의해 시작된 일괄처리를 되살리려고 하지 않기때문이다.

조작할 CD가 없다면 비공개함수 createNewRecord()를 호출하여 자료기지에 빈 CD를 삽입한다. 이것은 자리길들의 QDataTable에서 CD ID를 외부열쇠로 사용하게 한다. 사용자들이 Cancel을 찰칵하면 일괄처리를 되돌리고 빈 레코드는 표시되지 않는다.

이 대화칸에서는 ArtistForm에서와는 다른 자료기지련결을 사용한다. 그것은 련결당 오직 하나의 능동일괄처리를 가질수 있으므로 두 일괄처리를 요구하는 상황에서, 례를 들면 사용자가 Add New를 찰칵하여 ArtistForm을 펼치는 경우에 완료할수 있기때문이다.

```
titleLabel = new QLabel(tr("&Title:"), this);
artistLabel = new QLabel(tr("&Artist:"), this);
yearLabel = new QLabel(tr("&Year:"), this);
titleLineEdit = new QLineEdit(this);
yearSpinBox = new QSpinBox(this);
yearSpinBox->setRange(1900, 2100);
yearSpinBox->setValue(QDate::currentDate().year());
artistComboBox = new ArtistComboBox(db, this);
artistButton = new QPushButton(tr("Add &New..."), this);
...
cancelButton = new QPushButton(tr("Cancel"), this);
```

사용자대면부를 형성하는 표식자들과 행편집칸, 스핀칸, 복합칸, 단추들을 창조한다. 복합칸은 후에 설명하는 ArtistComboBox형이다.

```
trackCursor = new QSqlCursor("track", true, db);
trackTable = new QDataTable(trackCursor, false, this);
trackTable->setFilter("cdid = " + QString::number(cdId));
trackTable->setSort(trackCursor->index("number"));
trackTable->addColumn("title", tr("Track"));
trackTable->addColumn("duration", tr("Duration"));
trackTable->refresh();
```

사용자가 현재CD의 자리길들을 열람 및 편집하게 하는 QDataTable을 설정한다. 이것은 앞 절에서 ArtistForm클라스에 대하여 수행한것과 아주 비슷하다.

```
cdCursor = new QSqlCursor("cd", true, db);
cdCursor->select("id = " + QString::number(cdId));
```

```
cdCursor->next();
```

QSqlForm과 련관된 QSqlCursor를 설정하고 그것이 현재ID를 가지는 레코드를 지적하게 한다.

```
QSqlPropertyMap *propertyMap = new QSqlPropertyMap;
propertyMap->insert("ArtistComboBox", "artistId");
sqlForm = new QSqlForm(this);
sqlForm->installPropertyMap(propertyMap);
sqlForm->setRecord(cdCursor->primeUpdate());
sqlForm->insert(titleLineEdit, "title");
sqlForm->insert(artistComboBox, "artistid");
sqlForm->insert(yearSpinBox, "year");
sqlForm->readFields();
```

QSqlPropertyMap를 창조한다. QSqlPropertyMap클라스는 QSqlForm에게 어느 Qt속성이 어떤 형의 편집기창문부품의 값을 보관하는가를 말해준다. 기정으로 이미 QSqlForm은 QLineEdit가 값을 text속성에 보관하고 QSpinBox는 값을 value속성에 보관한다는것을 알고있다. 그러나 ArtistComboBox와 같은 사용자정의창문부품들에 대해서는 전혀 모른다. 속성매프에 쌍("ArtistComboBox", "artistId")을 삽입하고 QSqlForm에 대하여 installPropertyMap()을 호출함으로써 QSqlForm에 ArtistComboBox형의 창문부품들에서 artistId속성을 사용한다는것을 알린다.

또한 QSqlForm객체는 조작할 완충기를 요구하며 완충기는 QSqlCursor에 대하여 primeUpdate()를 호출하여 얻어지며 어느 편집기창문부품이 어느 자료기지마당에 대응하는가를 알아야 한다. 끝으로 readFields()를 호출하여 자료기지로부터 편집기창문부품들에 자료를 읽어들인다.

```
    connect(artistButton, SIGNAL(clicked()), this, SLOT(addNewArtist()));
    connect(moveUpButton, SIGNAL(clicked()), this, SLOT(moveTrackUp()));
    connect(moveDownButton, SIGNAL(clicked()), this, SLOT(moveTrackDown()));
    connect(updateButton, SIGNAL(clicked()), this, SLOT(accept()));
    connect(cancelButton, SIGNAL(clicked()), this, SLOT(reject()));
    connect(trackTable, SIGNAL(beforeInsert(QSqlRecord *)), this,
            SLOT(beforeInsertTrack(QSqlRecord *)));
    ...
}
```

단추들의 clicked()신호들과 QDataTable의 beforeInsert()신호를 다음에 서술하는 비공개처리부들에 련결한다.

```
void CdForm::accept()
{
```

```
        sqlForm->writeFields();
        cdCursor->update();
        db->commit();
        QDialog::accept();
    }
```

사용자가 Update를 찰칵하면 자료를 QSqlCursor의 편집완충기에 써넣고 update()를 호출하여 자료기지에 대하여 UPDATE를 수행하고 commit()를 호출하여 실제로 레코드를 자료기지에 써넣으며 기초클라스의 accept()실현을 호출하여 폼을 닫는다.

```
    void CdForm::reject()
    {
        db->rollback();
        QDialog::reject();
    }
```

사용자가 Cancel을 찰칵하면 되돌아와서 자료기지를 변경하지 않은대로 놔두고 폼을 닫는다.

```
    void CdForm::addNewArtist()
    {
        ArtistForm form(this);
        if (form.exec()) {
            artistComboBox->refresh();
            updateButton->setEnabled(artistComboBox->count() > 0);
        }
    }
```

사용자가 Add New를 찰칵하면 이행금지ArtistForm대화칸을 펼친다. 대화칸은 사용자가 새 기사들을 추가하게 하고 또한 현존기사들을 편집 및 삭제하게 한다. 사용자가 Update를 찰칵하면 ArtistComboBox::refresh()를 호출하여 그 기사목록이 갱신되도록 한다.

새 CD를 기사이름이 없이 창조하지 않게 하려고 하므로 기사가 있는가 없는가에 따라 *Update*단추를 허용하거나 금지한다.

```
    void CdForm::beforeInsertTrack(QSqlRecord *buffer)
    {
        buffer->setValue("id", generateId("track", db));
        buffer->setValue("number", trackCursor->size() + 1);
        buffer->setValue("cdid", cdId);
    }
```

beforeInsertTrack()처리부는 QDataTable의beforeInsert()신호에 련결된다. 레코드의 id, 번호,

그리고 cdid마당들을 설정한다.

```
void CdForm::beforeDeleteTrack(QSqlRecord *buffer)
{
    QSqlQuery query(db);
    query.exec("UPDATE track SET number = number -1 WHERE track.number > "
            + buffer->value("number").toString());
}
```

beforeDeleteTrack()처리부는 QDataTable의beforeDelete()신호에 련결된다. 삭제한 자리길보다 큰 수를 가지는 모든 자리길들의 번호를 다시 지정하여 자리길번호들이 련결되도록 한다. 례를 들면 CD에 6개의 자리길이 있고 사용자가 자리길4를 삭제하면 자리길5는 자리길4로 되고 자리길6은 자리길5로 된다.

아직 설명하지 않은 4개의 함수 moveTrackUp(), moveTrackDown(), swapTracks(), createNewRecord()가 있다. 이 함수들은 응용프로그람을 사용할수 있게 하는데 필요하지만 그 실현은 어떤 새로운 기술도 보여주지 않으므로 여기서 고찰하지 않는다.

이제는 CD Collection응용프로그람의 모든 코드를 고찰하였으므로 사용자정의 ArtistComboBox를 고찰할 준비가 되였다. 보통처럼 클라스정의부터 고찰하자.

```
class ArtistComboBox : public QComboBox
{
    Q_OBJECT
    Q_PROPERTY(int artistId READ artistId WRITE setArtistId)
public:
    ArtistComboBox(QSqlDatabase *database, QWidget *parent = 0, const char *name = 0);
    void refresh();
    int artistId() const;
    void setArtistId(int id);
private:
    void populate();
    QSqlDatabase *db;
    QMap<int, int> idFromIndex;
    QMap<int, int> indexFromId;
};
```

ArtistComboBox클라스는 QComboBox를 계승하며 artistId속성과 일부 함수들을 추가한다.

비공개절에서는 기사ID들을 복합칸첨수들과 련결하는 QMap<int, int>와 복합칸첨수들을 기사ID들과 련결하는 QMap<int, int>를 선언한다.

```
ArtistComboBox::ArtistComboBox(QSqlDatabase *database, QWidget *parent, const char *name)
```

```
    : QComboBox(parent, name)
{
    db = database;
    populate();
}
```

구성자에서는 비공개함수 populate()를 호출하여 복합칸에 기사표의 이름과 ID들을 채운다.

```
void ArtistComboBox::refresh()
{
    int oldArtistId = artistId();
    clear();
    idFromIndex.clear();
    indexFromId.clear();
    populate();
    setArtistId(oldArtistId);
}
```

refresh()함수에서는 복합칸에 자료기지의 최근자료를 다시 채운다. 또한 갱신전에 선택되였던 기사가 자료기지로부터 삭제되지 않았으면 후에도 계속 선택되여 있도록 하는데 주의해야 한다.

```
void ArtistComboBox::populate()
{
    QSqlCursor cursor("artist", true, db);
    cursor.select(cursor.index("name"));
    int index = 0;
    while (cursor.next()) {
        int id = cursor.value("id").toInt();
        insertItem(cursor.value("name").toString(), index);
        idFromIndex[index] = id;
        indexFromId[id] = index;
        ++index;
    }
}
```

비공개함수 populate()에서는 모든 기사들을 순환하면서 QComboBox::insertItem()를 호출하여 기사들을 복합칸에 추가한다. 또한 idFromIndex와 indexFromId매프들을 갱신한다.

```
int ArtistComboBox::artistId() const
```

```
{
    return idFromIndex[currentItem()];
}
```

artistId()함수는 현재 기사의 ID를 돌려준다.

```
void ArtistComboBox::setArtistId(int id)
{
    if (indexFromId.contains(id))
        setCurrentItem(indexFromId[id]);
}
```

setArtistId()함수는 기사ID에 기초하여 현재기사를 설정한다.

흔히 외부열쇠들을 표시하는 복합칸을 요구하는 응용프로그람들에서는 구성자가 표이름, 현시할 마당, ID들에 사용할 마당을 지적하게 하는 일반DatabaseComboBox클라스를 창조할 때 의의가 있다.

createConnections()와 main()함수를 실현하는것으로 CD Collection응용프로그람을 끝낸다.

```
inline bool createOneConnection(const QString &name)
{
    QSqlDatabase *db;
    if (name.isEmpty())
        db = QSqlDatabase::addDatabase("QSQLITEX");
    else
        db = QSqlDatabase::addDatabase("QSQLITEX", name);
    db->setDatabaseName("cdcollection.dat");
    if (!db->open()) {
        db->lastError().showMessage();
        return false;
    }
    return true;
}
inline bool createConnections()
{
    return createOneConnection("") && createOneConnection("ARTIST")
                    && createOneConnection("CD");
}
```

createConnections()에서는 CD자료기지에 대한 3개의 등가한 련결을 창조한다. 첫째 련결에는 이름을 주지 않고 기정으로 자료기지를 지정하지 않을 때 사용한다. 다른 련결들은

ARTIST와 CD라고 부르고 ArtistForm와 CdForm이 사용한다.

```
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    if (!createConnections())
        return 1;
    MainForm mainForm;
    app.setMainWidget(&mainForm);
    mainForm.resize(480, 320);
    mainForm.show();
    return app.exec();
}
```

main()함수는 다른 대부분의 Qt main()함수들과 비슷한데 createConnections()호출이 추가된다.

앞 절의 마감에 언급한것처럼 한가지 가능한 개선은 매개 자리길의 연주시간을 초가 아니라 분과 초로 표시하는것이다. 또한 QSqlCursor::calculateField()을 재정의함으로써 QSqlEditorFactory의 파생클라스를 만들어 사용자정의편집기(QTimeEdit에 기초할수 있다)를 제공하며 QSqlPropertyMap를 리용하여 QDataTable에게 편집기로부터 값을 어떻게 얻는가 하는것을 알려준다.(자세한 정보는 QDataTable의 installEditorFactory()와 installPropertyMap()함수들의 방조를 참고하시오.)

또 하나의 개선은 자료기지에서 매개 CD의 표지화상을 보관하고 CdForm에 그것을 표시하는것이다. 이것을 실현하려면 자료기지에서 화상자료를 BLOB로서 보관하며 그것을 QByteArray로서 얻어서 QByteArray를 QImage구성자에 넘긴다.

제13장. 망프로그람작성

Qt는 FTP 및 HTTP와 작업하기 위한 QFtp와 QHttp클라스를 제공한다. 이 통신규약들은 파일들을 내리적재 및 올리적재하는데 사용하기 쉬우며 HTTP의 경우에 웨브봉사기들에 요구를 보내고 결과를 얻는데 사용하기 편리하다.

Qt의 QFtp와 QHttp클라스들은 TCP소케트를 제공하는 저수준QSocket클라스에 구축된다. TCP는 망마디들사이에 전송되는 자료흐름에 의하여 조작된다. 또한 QSocket는 가동환경에 고유한 망API와 가까운 QSocketDevice에 실현된다. QSocketDevice클라스는 TCP와 UDP를 둘다 유지한다.

이 장에서는 우에서 언급한 4개의 클라스들과 QServerSocket, QSocketNotifier와 같이 밀접히 련관된 다른 클라스들의 사용법을 학습한다. 또한 파일들의 올리적재와 내리적재를 고찰하고 웨브폼을 프로그람적으로 사용하는 방법을 고찰한다. 봉사기프로그람과 대응하는 의뢰기프로그람에서 TCP를 사용한다. 마찬가지로 송신프로그람과 대응하는 수신프로그람에서 UDP를 사용한다. QFtp와 QHttp의 적용범위는 누구나 호출할수 있어야 하지만 QSocket와 특히 QSocketDevice의 적용범위는 망프로그람작성실험을 가정한다.

제1절. QFtp의 리용

QFtp클라스는 Qt에서 FTP통신규약의 의뢰기측을 실현한다. 이 클라스는 get(), put(), remove(), mkdir()를 비롯한 가장 일반적인 FTP조작들을 수행하기 위한 여러가지 함수들을 제공하며 임의의 FTP지령들을 실행하는 수단을 제공한다.

QFtp클라스는 비동기적으로 작업한다. get()나 put()와 같은 함수를 호출할 때 곧 되돌아오며 Qt의 사건순환고리로 조종이 넘어올 때 자료전송이 발생한다. 이것은 FTP지령들을 실행하는동안 사용자대면부에 응답성이 유지된다는것을 담보한다.

get()에 의해 하나의 파일을 얻는 방법을 보여주는 실례로 시작한다. 실례는 응용프로그람의 MainWindow클라스가 FTP싸이트로부터 값목록을 얻는데 필요하다고 가정한다.

```
class MainWindow : public QMainWindow
{
    Q_OBJECT
public:
    MainWindow(QWidget *parent = 0, const char *name = 0);
    void getPriceList();
    ...
private slots:
    void ftpDone(bool error);
private:
```

```
    QFtp ftp;
    QFile file;
    ...
};
```

이 클래스는 값목록파일을 얻는 공개함수 getPriceList()와 파일전송이 완료될 때 호출되는 비공개처리부 ftpDone(bool)을 가진다. 또한 이 클래스는 2개의 비공개변수를 가진다. ftp변수는 QFtp형으로서 FTP봉사기에로의 련결을 밀봉하고 file변수는 내리적재한 파일을 디스크에 써넣는데 쓰인다.

```
MainWindow::MainWindow(QWidget *parent, const char *name) : QMainWindow(parent, name)
{
    ...
    connect(&ftp, SIGNAL(done(bool)), this, SLOT(ftpDone(bool)));
}
```

구성자에서는 QFtp객체의 done(bool)신호를 ftpDone(bool)비공개처리부에 련결한다. QFtp는 모든 요구에 대한 처리가 끝났을 때 done(bool)신호를 발생한다. bool파라메터는 오유가 있는가 없는가를 가리킨다.

```
void MainWindow::getPriceList()
{
    file.setName("price-list.csv");
    if (!file.open(IO_WriteOnly)) {
        QMessageBox::warning(this, tr("Sales Pro"),
          tr("Cannot write file %1\n%2.") .arg(file.name()) .arg(file.errorString()));
        return;
    }
    ftp.connectToHost("ftp.trolltech.com");
    ftp.login();
    ftp.cd("/topsecret/csv");
    ftp.get("price-list.csv", &file);
    ftp.close();
}
```

getPriceList()함수는 ftp://ftp.trolltech.com/topsecret/csv/price-list.csv파일을 내리적재하여 현재 등록부의 price-list.csv로서 보관한다.

우선 써넣으려는 QFile을 연다. 그다음 QFtp객체를 리용하여 5개의 FTP지령을 차례로 실행한다. get()의 둘째 인수는 출력장치를 지정한다.

FTP지령들은 Qt의 사건순환고리에서 대기하고있다가 실행된다. 지령의 완료는 구성자에

서 ftpDone(bool)에 련결된 QFtp의 done(bool)신호에 의하여 알려진다.

```
void MainWindow::ftpDone(bool error)
{
    if (error)
        QMessageBox::warning(this, tr("Sales Pro"),
                tr("Error while retrieving file with FTP: %1.") .arg(ftp.errorString()));
    file.close();
}
```

일단 FTP지령들이 실행되면 파일을 닫는다. 오유가 발생하면 그것을 QMessageBox에 현시한다.

QFtp는 다음의 조작 즉 connectToHost(), login(), close(), list(), cd(), get(), put(), remove(), mkdir(), rmdir(), 그리고 rename()를 제공한다. 이 함수들은 모두 FTP지령을 발생하고 지령을 식별하는 ID번호를 돌려준다. 임의의 FTP지령들은 rawCommand()에 의해 실행될수 있다. 례를 들면 여기에 SITE CHMOD지령을 실행하는 방법이 있다.

```
ftp.rawCommand("SITE CHMOD 755 fortune");
```

QFtp는 지령실행을 시작할 때 commandStarted(int)신호를 발생하고 지령이 끝날 때 commandFinished(int, bool)신호를 발생한다. int파라메터는 지령을 식별하는 ID번호이다. 개별적인 지령들의 수명에 관심이 있으면 지령들을 실행할 때 ID번호들을 보관할수 있다. ID번호를 리용하여 사용자에게 지령에 대한 세부적인 처리를 제공할수 있다. 례를 들면

```
void MainWindow::getPriceList()
{
    ...
    connectId = ftp.connectToHost("ftp.trolltech.com");
    loginId = ftp.login();
    cdId = ftp.cd("/topsecret/csv");
    getId = ftp.get("price-list.csv", &file);
    closeId = ftp.close();
}
void MainWindow::commandStarted(int id)
{
    if (id == connectId) {
        statusBar()->message(tr("Connecting..."));
    } else if (id == loginId) {
        statusBar()->message(tr("Logging in..."));
        ...
```

```
}
```

반결합을 제공하는 다른 하나의 수법은 QFtp의 stateChanged()신호에 련결하는것이다.

대부분의 응용프로그람들에서는 오직 지령들의 전체 렬에 관심을 가진다. 그때 지령기다림렬이 빌 때마다 발생되는 done(bool)신호를 단순히 련결할수 있다.

오유가 발생하면 QFtp는 자동적으로 지령기다림렬을 지운다. 이것은 련결이나 로그인이 실패하면 기다림렬의 뒤를 따르는 지령들은 절대로 실행되지 않는다는것을 의미한다. 그러나 오유발생후에 같은 QFtp객체를 리용하여 새 지령들을 실행한다면 이 지령들은 아무것도 발생하지 않은것처럼 기다렸다가 실행된다.

그러면 더 고급한 실례를 고찰하자.

```
class Downloader : public QObject
{
    Q_OBJECT
public:
    Downloader(const QUrl &url);
signals:
    void finished();
private slots:
    void ftpDone(bool error);
    void listInfo(const QUrlInfo &urlInfo);
private:
    QFtp ftp;
    std::vector<QFile *> openedFiles;
};
```

Downloader클라스는 FTP등록부에 배치되는 모든 파일들을 내리적재한다. 등록부는 클라스의 구성자에 넘긴 QUrl로서 지정된다. QUrl클라스는 파일이름, 경로, 통신규약 및 포구와 같은 URL의 각 부분들을 꺼내기 위한 고급한 대면부를 제공하는 Qt클라스이다.

```
Downloader::Downloader(const QUrl &url)
{
    if (url.protocol() != "ftp") {
        QMessageBox::warning(0, tr("Downloader"), tr("Protocol must be 'ftp'."));
        emit finished();
        return;
    }
    int port = 21;
    if (url.hasPort())
```

```
        port = url.port();
    connect(&ftp, SIGNAL(done(bool)), this, SLOT(ftpDone(bool)));
    connect(&ftp, SIGNAL(listInfo(const QUrlInfo &)), this, SLOT(listInfo(const QUrlInfo &)));
    ftp.connectToHost(url.host(), port);
    ftp.login(url.user(), url.password());
    ftp.cd(url.path());
    ftp.list();
}
```

구성자에서는 우선 URL이 "ftp:"로 시작하는가 검사한다. 그다음 포구번호를 꺼낸다. 포구가 지정되지 않으면 FTP의 기정포구인 포구 21을 사용한다.

다음으로 2개의 신호–처리부련결을 확립하고 4개의 FTP지령들을 실행한다. 마지막 FTP지령 list()는 등록부안의 매개 파일이름을 얻으며 얻어지는 매개 이름에 대하여 listInfo(const QUrlInfo &)신호를 발생한다. 이 신호는 주어진 URL과 련관된 파일을 내리적재하는 listInfo()라고 부르는 처리부에 련결된다.

```
void Downloader::listInfo(const QUrlInfo &urlInfo)
{
    if (urlInfo.isFile() && urlInfo.isReadable()) {
        QFile *file = new QFile(urlInfo.name());
        if (!file->open(IO_WriteOnly)) {
            QMessageBox::warning(0, tr("Downloader"), tr("Error: Cannot open file %1:\n%2.")
                .arg(file->name()) .arg(file->errorString()));
            emit finished();
            return;
        }
        ftp.get(urlInfo.name(), file);openedFiles.push_back(file);
    }
}
```

listInfo()처리부의 QUrlInfo파라메터는 원격파일에 대한 자세한 정보를 제공한다. 파일이 표준파일(등록부아님)이고 읽기가능하다면 get()를 호출하여 그것을 내리적재한다. 내리적재에 사용된 QFile객체는 new에 의하여 할당되고 그 지적자는 openedFiles벡토르에 보관된다.

```
void Downloader::ftpDone(bool error)
{
    if (error)
        QMessageBox::warning(0, tr("Downloader"), tr("Error: %1.") .arg(ftp.errorString()));
    for (int i = 0;i < (int)openedFiles.size();++i)
```

```
        delete openedFiles[i];
    emit finished();
}
```

ftpDone()처리부는 FTP지령들이 모두 완료하였을 때 혹은 오유가 발생하면 호출된다. QFile객체들을 삭제하여 기억루실을 방지하고 또한 매개 파일을 닫는다. (QFile해체자는 파일이 열려있으면 자동적으로 닫는다.)

오유가 없으면 FTP지령들과 신호들의 렬은 다음과 같다.

```
connectToHost(host)
login()
cd(path)
list()
    emit listInfo(file_1)
        get(file_1)
    emit listInfo(file_2)
        get(file_2)
    ...
    emit listInfo(file_N)
        get(file_N)
emit done()
```

가령 12개 파일중 5번째 파일을 내리적재하는동안 망오유가 발생하면 남은 파일들은 내리적재되지 않는다. 될수록 많은 파일들을 내리적재하려고 한다면 하나의 대책은 한번에 하나씩 GET조작을 실행하고 새로운 GET조작을 실행하기전에 done(bool)신호를 기다리는것이다. listInfo()에서는 get()를 호출하는것이 아니라 단순히 파일이름을 QStringList에 추가하고 done(bool)에서 다음 파일에 대하여 get()를 호출하여 QStringList에 내리적재한다. 그때 실행순서는 다음과 같다.

```
connectToHost(host)
login()
cd(path)
list()
    emit listInfo(file_1)
    emit listInfo(file_2)
    ...
    emit listInfo(file_N)
emit done()
get(file_1)
```

```
emit done()
get(file_2)
emit done()
...
get(file_N)
emit done()
```

다른 대책은 파일당 하나의 QFtp객체를 사용하는것이다. 이것은 개별적인 FTP련결들을 통하여 파일들을 병렬로 내리적재하게 한다.

```
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    QUrl url("ftp://ftp.example.com/");
    if (argc >= 2)
        url = argv[1];
    Downloader downloader(url);
    QObject::connect(&downloader, SIGNAL(finished()),&app, SLOT(quit()));
    return app.exec();
}
```

main()함수로 프로그람을 끝낸다. 사용자가 지령행에서 URL을 지정하면 그것을 사용하고 그렇지 않으면 ftp://ftp.example.com/에로 간다.

두 실례에서 get()로 얻은 자료는 QFile에 써넣어진다. 사실은 그렇지 않은 경우도 있다. 기억기안의 자료가 요구된다면 QByteArray를 포함하는 QIODevice의 파생클라스인 QBuffer를 사용할수 있다. 례를 들면

```
QBuffer *buffer = new QBuffer(byteArray);
buffer->open(IO_WriteOnly);
ftp.get(urlInfo.name(), buffer);
```

또한 get()에 대한 입출력인수를 생략하거나 null지적자를 넘길수도 있다. 그때 QFtp클라스는 새 자료를 얻을 때마다 readyRead()신호를 발생하고 자료는 readBlock()나 readAll()에 의하여 읽어들일수 있다.

자료를 내리적재하는동안 사용자에게 반결합을 제공하려고 한다면 QFtp의 dataTransferProgress(int, int)신호를 QProgressBar 혹은 QProgressDialog의 setProgress(int, int)처리부에 련결한다. 또한 QProgressBar나 QProgressDialog의 canceled()신호를 QFtp의 abort()처리부에 련결한다.

제2절. QHttp의 리용

QHttp클라스는 Qt에서 HTTP통신규약의 의뢰기측을 실현한다. 이 클라스는 get()와 post()

를 비롯한 가장 일반적인 HTTP조작들을 수행하는 각종 함수들을 제공하고 임의의 HTTP요구를 보내는 수단을 제공한다. 앞 절에서 QFtp와 QHttp사이에 류사성이 많다는것을 보았다.

QHttp클라스는 비동기적으로 작업한다. get()나 post()와 같은 함수를 호출할 때 함수는 곧 되돌아오고 후에 조종이 Qt의 사건순환고리에 돌아올 때 자료전송이 발생한다. 이것은 HTTP요구를 처리하는동안에 응용프로그람의 사용자대면부가 응답성을 유지하도록 한다.

Qt응용프로그람의 MainWindow클라스에서 Trolltech의 웨브싸이트로부터 HTML파일을 내리적재하는 방법을 보여주는 실례를 고찰한다. 머리부파일은 앞 절에서 사용한것과 거의 비슷하므로 생략하는데 하나의 비공개처리부(httpDone(bool))와 비공개변수들(QHttp형의 http와 QFile형의 file)이 있다.

```
MainWindow::MainWindow(QWidget *parent, const char *name) : QMainWindow(parent, name)
{
    ...
    connect(&http, SIGNAL(done(bool)), this, SLOT(httpDone(bool)));
}
```

구성자에서는 QHttp객체의 done(bool)신호를 MainWindow의 httpDone(bool)처리부에 련결한다.

```
void MainWindow::getFile()
{
    file.setName("aboutqt.html");
    if (!file.open(IO_WriteOnly)) {
        QMessageBox::warning(this, tr("HTTP Get"), tr("Cannot write file %1\n%2.")
            .arg(file.name())
            .arg(file.errorString()));
        return;
    }
    http.setHost("doc.trolltech.com");
    http.get("/3.2/aboutqt.html", &file);
    http.closeConnection();
}
```

getFile()함수는 http://doc.trolltech.com/3.2/aboutqt.html파일을 내리적재하여 현재등록부에 aboutqt.html로서 보관한다.

QFile을 써넣기용으로 열고 QHttp객체를 리용하여 3개의 HTTP요구들의 렬을 실행한다. get()의 둘째 인수는 출력장치를 지정한다.

HTTP요구들은 Qt의 사건순환고리에서 기다렸다가 실행된다. 지령의 완료는 구성자에서 httpDone(bool)에 련결된 QHttp의 done(bool)신호에 의해 지적된다.

```
void MainWindow::httpDone(bool error)
{
    if (error)
        QMessageBox::warning(this, tr("HTTP Get"),
            tr("Error while fetching file with HTTP: %1.") .arg(http.errorString()));
    file.close();
}
```

일단 HTTP요구들이 완료되면 파일을 닫는다. 오유가 발생하였다면 QMessageBox에 오유통보문을 현시한다.

QHttp는 다음의 조작 즉 setHost(), get(), post(), head()를 제공한다. 례를 들면 여기에 post()를 리용하여 "*name = value*"쌍들의 목록을 CGI스크립트에 보내는 방법이 있다.

```
http.setHost("www.example.com");
http.post("/cgi/somescript.py", QCString("x=200&y=320"), &file);
```

많은 조종들에서 임의의 HTTP머리부와 자료를 받아들이는 request()함수를 사용할수 있다. 례를 들면

```
QHttpRequestHeader header("POST", "/search.html");
header.setValue("Host", "www.trolltech.com");
header.setContentType("application/x-www-form-urlencoded");
http.setHost("www.trolltech.com");
http.request(header, QCString("qt-interest=on&search=opengl"));
```

QHttp는 요구의 실행을 시작할 때 requestStarted(int)신호를 발생하고 요구가 완료되였을 때 requestFinished(int, bool)신호를 발생한다. int파라메터는 요구를 식별하는 ID번호이다. 개별적인 요구들의 수명에 관심이 있다면 요구들을 실행할 때 ID번호들을 보관할수 있다. ID번호를 리용하여 사용자에게 요구에 대한 세부적인 조종을 제공할수 있다.

대부분의 응용프로그람들에서는 요구들의 전체렬이 성과적으로 끝났는가 아닌가를 알려고만 한다. 이것은 요구기다림렬이 비게 될 때 발생되는 done(bool)신호에 련결하여 간단히 달성한다.

오유가 발생하면 요구는 자동적으로 지워진다. 그러나 오유발생후에 같은 QHttp객체를 리용하여 새로운 요구를 실행한다면 이 요구들은 보통과 같이 기다렸다가 전송된다.

QFtp처럼 QHttp는 입출력장치를 지정할 대신에 사용할수 있는 readBlock()와 readAll()함수들은 물론 readyRead()신호를 제공한다. 또한 QProgressBar 혹은 QProgressDialog의 setProgress(int, int)처리부에 련결할수 있는 dataTransferProgress(int, int)신호를 제공한다.

제3절. QSocket를 리용한 TCP망프로그람작성

QSocket클라스는 TCP의뢰기와 봉사기들을 실현하는데 쓰일수 있다. TCP는 FTP와 HTTP를 비롯한 수많은 응용층Internet통신규약들의 기초를 이루는 전송층통신규약으로서 전용통신

규약들에도 사용할수 있다.

TCP는 흐름지향통신규약이다. 응용프로그람들에서 자료는 크고 평탄한 파일보다도 긴 흐름으로 나타난다. TCP상에서 구축된 웃준위통신규약들은 일반적으로 행지향 혹은 블로크지향이다.

· 행지향통신규약들은 자료를 행바꾸기로 구분한 본문행으로 전송한다.

· 블로크지향통신규약들은 자료를 2진자료블로트들로 전송하다. 매개 블로크는 크기마당과 그뒤의 자료의 크기byte로 구성된다.

QSocket는 QIODevice를 계승하므로 QDataStream 혹은 QTextStream을 리용하여 읽고쓸수 있다. 망으로부터 자료를 읽어들일 때 파일로부터 읽어들일 때와 다른 한가지 중요한 차이는 >>연산자를 사용하기전에 동료로부터 충분한 자료를 받았다는것을 확인해야 하는것이다. 여기서의 실패는 정의되지 않은 동작을 발생시킬수 있다.

이 절에서는 전용블로크지향통신규약을 사용하는 의뢰기와 봉사기의 코드를 고찰한다. 의뢰기는 Trip Planner라고 부르고 사용자들이 다음번 렬차려행을 계획하게 한다. 봉사기는 Trip Server라고 부르며 의뢰기에 려행정보를 제공한다. Trip Planner응용프로그람을 쓰는것으로 시작한다.

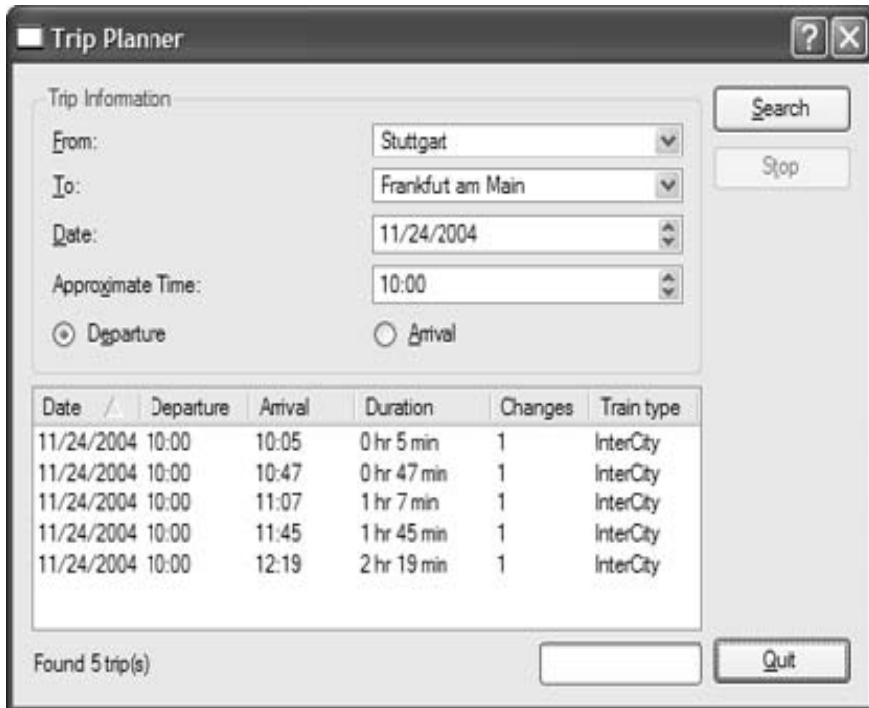



그림 13-1. Trip Planner응용프로그람

Trip Planner는 From마당, To마당, Date마당, Approximate Time마당 각각 하나씩과 근사시간이 출발시간인가 도착시간인가를 선택하는 2개의 라지오단추들을 제공한다. 사용자가 Search를 찰칵할 때 응용프로그람은 요구를 봉사기에 보내고 봉사기는 사용자의 기준에 맞는 렬차려행들의 목록으로 응답한다. 목록은 Trip Planner창문의 QListView에 표시된다. 창문의 제일 아래

에는 마지막조작의 상태를 보여주는 QLabel과 QProgressBar이 있다.

Trip Planner의 사용자대면부는 Qt Designer로 창조한다. 여기서는 대응하는 .ui.h파일의 원천코드에 초점을 둔다. 다음의 4개 변수를 Qt Designer의 Members타브에서 선언한다.

```
QSocket socket;
QTimer connectionTimer;
QTimer progressBarTimer;
Q_UINT16 blockSize;
```

QSocket형의 socket변수는 TCP련결을 밀봉한다. connectionTimer변수는 오래 지속되는 련결시간을 요구하는데 쓰인다. progressBarTimer변수는 응용프로그람이 작업중에 있을 때 진척상황띠를 주기적으로 갱신하는데 쓰인다. 끝으로 blockSize변수는 봉사기로부터 받은 블로크들을 해석할 때 사용된다.

```
void TripPlanner::init()
{
    connect(&socket, SIGNAL(connected()), this, SLOT(sendRequest()));
    connect(&socket, SIGNAL(connectionClosed()), this, SLOT(connectionClosedByServer()));
    connect(&socket, SIGNAL(readyRead()), this, SLOT(updateListView()));
    connect(&socket, SIGNAL(error(int)), this, SLOT(error(int)));
    connect(&connectionTimer, SIGNAL(timeout()), this, SLOT(connectionTimeout()));
    connect(&progressBarTimer, SIGNAL(timeout()), this, SLOT(advanceProgressBar()));
    QDateTime dateTime = QDateTime::currentDateTime();
    dateEdit->setDate(dateTime.date());
    timeEdit->setTime(QTime(dateTime.time().hour(), 0));
}
```

init()에서는 QSocket의 connected(), connectionClosed(), readyRead(), error(int)신호들과 두개 시계들의 timeout()신호들을 자체의 처리부들에 련결한다. 또한 Date와 Approximate Time마당들에 현재날자와 시간에 기초한 기정값들을 채워넣는다.

```
void TripPlanner::advanceProgressBar()
{
     progressBar->setProgress(progressBar->progress() + 2);
}
```

advanceProgressBar()처리부는 progressBarTimer의 timeout()신호에 련결된다.

진척상황띠를 2단위 전진시킨다. Qt Designer에서 진척상황띠의 totalSteps속성을 띠가 작업중지시기로서 동작해야 한다는것을 의미하는 특수값인 0으로 설정한다.

```
void TripPlanner::connectToServer()
{
```

```
        listView->clear();
        socket.connectToHost("tripserver.zugbahn.de", 6178);
        searchButton->setEnabled(false);
        stopButton->setEnabled(true);
        statusLabel->setText(tr("Connecting to server..."));
        connectionTimer.start(30 * 1000, true);
        progressBarTimer.start(200, false);
        blockSize = 0;
    }
```

connectToServer()처리부는 사용자가 탐색을 시작하기 위하여 Search를 찰칵할 때 실행된다. QSocket객체에 대하여 connectToHost()를 호출하여 봉사기에 련결한다. 이때 가상주콤퓨터 tripserver.zugbahn.de를 포구6178에서 호출할수 있다고 가정한다. (자기의 콤퓨터에서 실례를 실행하려고 한다면 주콤퓨터이름을 국부주콤퓨터로 고친다.) connectToHost()호출은 비동기적이고 곧 되돌아온다. 일반적으로 련결은 후에 수립된다. QSocket객체는 련결이 이루어지고 실행될 때 connected()신호를 발생하고 련결이 실패하면 error(int) (오유코드와 함께)를 발생한다.

다음으로 사용자대면부를 갱신하고 2개의 시계를 기동한다. 첫째 시계 connectionTimer는 련결이 30s동안 이루어지지 않았을 때 절환되는 단일발사시계이다. 둘째 시계 progressBarTimer는 매번 200ms간격으로 사건을 발생하여 응용프로그람의 진척상황띠를 갱신하며 응용프로그람이 작업하고있다는 시각적인 암시를 사용자에게 준다.

끝으로 blockSize변수를 0으로 설정한다. blockSize변수는 봉사기로부터 받은 블로크의 길이를 보관한다. 다음에는 블로크크기를 아직 모른다는것을 의미하는 값 0을 사용하기로 선택하였다.

```
    void TripPlanner::sendRequest()
    {
        QByteArray block;
        QDataStream out(block, IO_WriteOnly);
        out.setVersion(5);
        out << (Q_UINT16)0 << (Q_UINT8) 'S'
            << fromComboBox->currentText()
            << toComboBox->currentText() << dateEdit->date()
            << timeEdit->time();
        if (departureRadioButton->isOn())
            out << (Q_UINT8) 'D';
        else
            out << (Q_UINT8) 'A';
```

```
    out.device()->at(0);
    out << (Q_UINT16) (block.size() -sizeof(Q_UINT16));
    socket.writeBlock(block.data(), block.size());
    statusLabel->setText(tr("Sending request..."));
}
```

sendRequest()처리부는 련결이 확립되였다는것을 가리키는 connected()신호를 QSocket객체가 발생할 때 실행된다. 그 처리부의 과제는 사용자가 입력한 모든 정보를 가지는 요구를 봉사기에 생성하는것이다.

요구는 다음 형식의 2진블로크이다.

Q_UINT16	이 마당을 제외한 블로크크기, byte수
Q_UINT8	요구형(늘 'S')
QString	출발도시
QString	도착도시
QDate	려행날자
QTime	근사려행시간
Q_UINT8	시간은 출발시간('D') 혹은 도착시간('A')이다.

우선 날자를 블로크라고 부르는 QByteArray에 써넣는다. 블로크안에 자료를 모두 넣은 다음에야 처음에 전송해야 할 블로크의 크기를 알수 있으므로 QSocket에 직접 자료를 써넣을수 있다.

초기에 블로크크기로서 0을 써넣고 다음에 자료의 나머지가 뒤에 온다. 그다음 입출력장치(배경에서 QDataStream에 의해 창조된 QBuffer)에 대하여 at(0)을 호출하여 byte배렬의 선두로 이동하고 블로크자료의 크기와 함께 0을 다시 써넣는다. 크기는 블로크크기에 sizeof(Q_UINT16) (즉 2)를 덜어서 byte량에서 크기마당을 제외한다. 그후에 QSocket에 대하여 writeBlock()를 호출하여 봉사기에 블로크를 보낸다.

```
void TripPlanner::updateListView()
{
    connectionTimer.start(30 * 1000, true);
    QDataStream in(&socket);
    in.setVersion(5);
    for (;;) {
        if (blockSize == 0) {
            if (socket.bytesAvailable() < sizeof(Q_UINT16))
                break;
```

```
            in >> blockSize;
        }
        if (blockSize == 0xFFFF) {
            closeConnection();
            statusLabel->setText(tr("Found %1 trip(s)") .arg(listView->childCount()));
            break;
        }
        if (socket.bytesAvailable() < blockSize)
            break;
        QDate date;
        QTime departureTime;
        QTime arrivalTime;
        Q_UINT16 duration;
        Q_UINT8 changes;
        QString trainType;
        in >> date >> departureTime >> duration >> changes >> trainType;
        arrivalTime = departureTime.addSecs(duration * 60);
        new QListViewItem(listView, date.toString(LocalDate),
              departureTime.toString(tr("hh:mm")), arrivalTime.toString(tr("hh:mm")),
              tr("%1 hr %2 min") .arg(duration / 60) .arg(duration % 60),
              QString::number(changes), trainType);
        blockSize = 0;
    }
}
```

updateListView()처리부는 QSocket가 봉사기로부터 새 자료를 수신할 때마다 발생되는 QSocket의 readyRead()신호와 련결된다. 처음으로 할 일은 단일발사련결시계를 재기동하는것이다. 봉사기로부터 자료를 수신할 때마다 련결이 살아있다는것을 알고있으므로 시계가 30s동안 실행되도록 설정한다.

봉사기는 사용자의 기준에 맞는 가능한 렬차려행목록을 송신한다. 그 매개 려행은 단일블로크로서 전송되고 매개 블로크는 크기로 시작된다. for순환의 코드가 복잡한것은 봉사기로부터 한번에 1개 자료블로크를 꼭 얻지 못하는데 있다. 블로크전체 혹은 블로크의 일부 혹은 1.5개블로크, 지어는 모든 블로크들을 한번에 수신할수 있다.

51 bytes | 48 bytes | 53 bytes
51 data | 48 data | ··· | 53 data | 0xFFFF

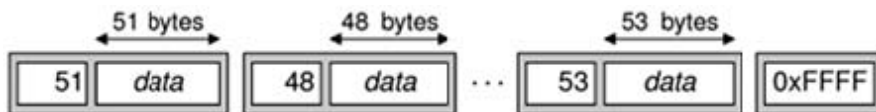



그림 13-2. Trip Server의 블로크들

그러면 for순환이 어떻게 작업하는가? blockSize변수가 0이면 이것은 다음 블로크의 크기를 읽지 못했다는것을 의미하므로 그것을 읽으려고 한다. (읽을수 있는것이 적어도 2byte 있다고 가정한다.) 봉사기는 값 0xFFFF를 사용하여 받을 자료가 더는 없다는것을 지정하므로 이 값을 읽어들이면 끝에 이르렀다는것을 알게 된다.

블로크크기가 0xFFFF아니면 다음 블로크를 읽으려고 시도한다. 우선 읽을수 있는 블로크크기byte가 있는가 알아보려고 한다. 없으면 거기서 중지한다. 유효자료가 있으면 readyRead() 신호가 다시 발생되고 그다음 다시 시도한다.

블로크전체가 도착했다는것이 확실하면 QSocket에 설정한 QDataStream에 대하여 >>연산자를 리용하여 려행과 관련한 정보를 안전하게 꺼내고 그 정보를 리용하여 QListViewItem을 창조한다. 봉사기로부터 수신한 블로크는 다음의 형식을 가진다.

Q_UINT16	이 마당을 제외한 블로크크기, byte수
QDate	출발날자
QTime	출발시간
Q_UINT16	운행시분
Q_UINT8	변경회수
QString	렬차형

끝으로 blockSize변수를 0으로 재설정하여 다음 블로크의 크기가 알려지지 않았고 읽어들여야 한다는것을 가리킨다.

```
void TripPlanner::closeConnection()
{
    socket.close();
    searchButton->setEnabled(true);
    stopButton->setEnabled(false);
    connectionTimer.stop();
    progressBarTimer.stop();
    progressBar->setProgress(0);
}
```

closeConnection()비공개함수는 TCP봉사기에로의 련결을 닫고 사용자대면부를 갱신하고 시계들을 정지시킨다. 이 함수는 0xFFFF를 읽어들일 때 updateListView()로부터 호출되며 우리가 간단히 설명하는 다른 처리부들로부터도 호출된다.

```
void TripPlanner::stopSearch()
{
```

```
        statusLabel->setText(tr("Search stopped"));
        closeConnection();
    }
```

stopSearch()처리부는 Stop단추의 clicked()신호에 련결된다. 본질상 이것은 closeConnection()를 호출한다.

```
    void TripPlanner::connectionTimeout()
    {
        statusLabel->setText(tr("Error: Connection timed out"));
        closeConnection();
    }
```

connectionTimeout()처리부는 connectionTimer의 timeout()신호에 련결된다.

```
    void TripPlanner::connectionClosedByServer()
    {
        if (blockSize != 0xFFFF)
            statusLabel->setText(tr("Error: Connection closed by server"));
        closeConnection();
    }
```

connectionClosedByServer()처리부는 소케트의 connectionClosed()신호에 련결된다.

봉사기가 련결을 닫고 아직 흐름끝표식기호인 0xFFFF를 수신하지 못했으면 사용자에게 오유가 발생하였다고 알린다. 보통처럼 closeConnection()를 호출하여 사용자대면부를 갱신하고 시계들을 정지시킨다.

```
    void TripPlanner::error(int code)
    {
        QString message;
        switch (code) {
            case QSocket::ErrConnectionRefused:
                message = tr("Error: Connection refused");
                break;
            case QSocket::ErrHostNotFound:
                message = tr("Error: Server not found");
                break;
            case QSocket::ErrSocketRead:
            default:
                message = tr("Error: Data transfer failed");
        }
```

```
        statusLabel->setText(message);
        closeConnection();
    }
```

error(int)처리부는 소케트의 error(int)신호에 련결되고 오유코드에 기초한 오유통보문를 생성한다.

Trip Planner응용프로그람의 main()함수는 우리가 기대한것과 같다.

```
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    TripPlanner tripPlanner;app.setMainWidget(&tripPlanner);
    tripPlanner.show();
    return app.exec();
}
```

그러면 봉사기를 실현하자. 봉사기는 2개의 클라스 즉 TripServer와 ClientSocket로 이루어진다. TripServer클라스는 들어오는 TCP련결을 받아들이는 클라스 QServerSocket를 계승한다. ClientSocket는 QSocket를 재정의하며 단일련결을 처리한다. 임의의 시간에 기억기에는 봉사받고있는 의뢰기로서 많은 ClientSocket객체가 있을수 있다.

```
class TripServer : public QServerSocket
{
public:
    TripServer(QObject *parent = 0, const char *name = 0);
    void newConnection(int socket);
};
```

TripServer클라스는 QServerSocket로부터 newConnection()함수를 재정의한다. 이 함수는 봉사기가 요구를 받으려는 포구에 의뢰기가 련결하려고 시도할 때마다 호출된다.

```
TripServer::TripServer(QObject *parent, const char *name)
        : QServerSocket(6178, 1, parent, name) {}
```

TripServer구성자에서는 포구번호(6178)를 기초클라스구성자에 넘긴다. 둘째 인수 1은 허용련결수이다.

```
void TripServer::newConnection(int socketId)
{
    ClientSocket *socket = new ClientSocket(this);
    socket->setSocket(socketId);
}
```

newConnection()에서는 ClientSocket객체를 TripServer객체의 자식으로 창조하고 그 소케트

ID를 주어진 수로 설정한다.

```
class ClientSocket : public QSocket
{
    Q_OBJECT
public:
    ClientSocket(QObject *parent = 0, const char *name = 0);
private slots:
    void readClient();
private:
    void generateRandomTrip(const QString &from, const QString &to, const QDate &date,
                    const QTime &time);
    Q_UINT16 blockSize;
};
```

ClientSocket클라스는 QSocket를 계승하며 단일의뢰기의 상태를 밀봉한다.

```
ClientSocket::ClientSocket(QObject *parent, const char *name) : QSocket(parent, name)
{
    connect(this, SIGNAL(readyRead()), this, SLOT(readClient()));
    connect(this, SIGNAL(connectionClosed()), this, SLOT(deleteLater()));
    connect(this, SIGNAL(delayedCloseFinished()), this, SLOT(deleteLater()));
    blockSize = 0;
}
```

구성자에서는 필요한 신호–처리부련결을 확립하고 blockSize변수를 0으로 설정하여 의뢰기로부터 보내온 블로크크기를 아직 모른다는것을 가리킨다.

connectionClosed()와 delayedCloseFinished()신호는 Qt의 사건순환고리에로 조종이 돌아올 때 객체를 삭제하는 QObject계승함수 deleteLater()에 련결된다. 이것은 동료에 의해 련결이 닫길 때 혹은 지연된 닫기를 완료할 때 ClientSocket객체가 삭제된다는것을 담보한다.

```
void ClientSocket::readClient()
{
    QDataStream in(this);
    in.setVersion(5);
    if (blockSize == 0) {
        if (bytesAvailable() < sizeof(Q_UINT16))
            return;
        in >> blockSize;
    }
```

```
        if (bytesAvailable() < blockSize)
            return;
        Q_UINT8 requestType;
        QString from;
        QString to;
        QDate date;
        QTime time;
        Q_UINT8 flag;
        in >> requestType;
        if (requestType == 'S') {
            in >> from >> to >> date >> time >> flag;
            srand(time.hour() * 60 + time.minute());
            int numTrips = rand() % 8;
            for (int i = 0;i < numTrips;++i)
                generateRandomTrip(from, to, date, time);
            QDataStream out(this);
            out << (Q_UINT16)0xFFFF;
        }
        close();
        if (state() == Idle)
            deleteLater();
    }
```

readClient()처리부는 QSocket의 readyRead()신호에 련결된다. blockSize가 0이면 blockSize를 읽는것으로 시작하고 그렇지 않으면 그것을 이미 읽었으므로 그대신에 전체 블로크가 도달했는가 검사한다. 전체 블로크를 읽을 준비가 되였으면 읽어들인다. QSocket(this객체)에 대하여 직접 QDataStream을 사용하며 >>연산자에 의하여 마당을 읽어들인다.

의뢰기의 요구를 읽었으면 응답을 생성할 준비가 된다. 이것이 실제현장에서 가동하는 응용프로그람이면 렬차시간표자료기지에서 정보를 검색하여 일치하는 렬차려행을 찾으려고 시도한다. 그러나 여기서는 우연적인 려행을 생성하는 generateRandomTrip()라는 함수로 충족시킨다. 이 함수를 우연회수 호출하고 0xFFFF를 보내여 자료의 끝을 지정한다.

끝으로 련결을 닫는다. 소케트의 출력완충기가 비였으면 련결은 곧 끝나고 조종이 Qt의 사건순환고리로 돌아올 때 deleteLater()를 호출하여 이 객체를 삭제한다. (이것은 delete this보다 더 안전하다.) 그렇지 않으면 QSocket는 모든 자료전송을 끝내고 련결을 닫으며 delayedCloseFinished()신호를 발생한다.

```
    void ClientSocket::generateRandomTrip(const QString &, const QString &, const QDate &date,
```

```
                const QTime &time)
{
    QByteArray block;
    QDataStream out(block, IO_WriteOnly);
    out.setVersion(5);
    Q_UINT16 duration = rand() % 200;
    out << (Q_UINT16)0 << date << time << duration
        << (Q_UINT8) 1 << QString("InterCity");
    out.device()->at(0);
    out << (Q_UINT16) (block.size() -sizeof(Q_UINT16));
    writeBlock(block.data(), block.size());
}
```

generateRandomTrip()함수는 TCP련결에서 자료블로크를 송신하는 방법을 보여준다. 이것은 의뢰기의 sendRequest()함수에서 수행한것과 거의 비슷하다. 다시 한번 블로크를 QByteArray에 써넣고 writeBlock()에 의해 송신하기전에 크기를 결정할수 있다.

```
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    TripServer server;
    if (!server.ok()) {
        qWarning("Failed to bind to port");
        return 1;
    }
    QPushButton quitButton(QObject::tr("&Quit"), 0);
    quitButton.setCaption(QObject::tr("Trip Server"));
    app.setMainWidget(&quitButton);
    QObject::connect(&quitButton, SIGNAL(clicked()), &app, SLOT(quit()));
    quitButton.show();
    return app.exec();
}
```

main()에서는 TripServer객체와 사용자가 봉사기를 정지시킬수 있는 QPushButton을 창조한다.

이로서 의뢰기–봉사기실례를 끝낸다. 이 경우에 읽고쓰기에 QDataStream을 사용하는 블로크지향통신규약을 사용하였다. 행지향통신규약을 사용하려고 하는 경우에 가장 간단한 수법은 readyRead()신호에 련결된 처리부에서 QSocket의 canReadLine()과 readLine()함수를 사용하는

것이다.

```
QStringList lines;
while (socket.canReadLine())
    lines.append(socket.readLine());
```

그다음 읽어들인 매 행을 처리한다. 자료전송에서와 같이 이것은 QSocket에서 QTextStream을 사용하여 수행한다.

우리가 사용한 봉사기실현은 련결이 많을 때에는 잘 동작하지 않는다. 문제는 요구를 처리하는동안 다른 련결을 처리하지 못하는데 있다. 더 합리적인 수법은 매개 련결에 대하여 새 스레드를 시작하는것이다. 그러나 QSocket는 오직 사건순환고리를 포함하는 스레드(QApplication::exec()호출)에서 사용할수 있다. 그 리유는 17장(다중스레드프로그람작성)에서 설명한다. 해결책은 사건순환고리에 의존하지 않는 저수준의 QSocketDevice클라스를 직접 사용하는것이다.

제4절. QSocketDevice를 리용한 UDP망프로그람작성

QSocketDevice클라스는 TCP와 UDP에서 사용할수 있는 저수준대면부를 제공한다. 대부분의 TCP응용프로그람들에서는 고수준QSocket클라스를 요구하지만 UDP를 사용하려면 직접 QSocketDevice를 사용해야 한다.

UDP는 믿음성없는 데이터그램지향통신규약이다. 일부 응용층통신규약들은 TCP보다 더 가벼운 UDP를 사용한다. UDP에서 자료는 한 주콤퓨터로부터 다른 주콤퓨터에로 파케트(데이터그램들)로서 송신된다. 거기에는 련결이란 개념이 없고 UDP파케트가 성공적으로 송달되지 않아도 체계에는 오유가 통보되지 않는다.

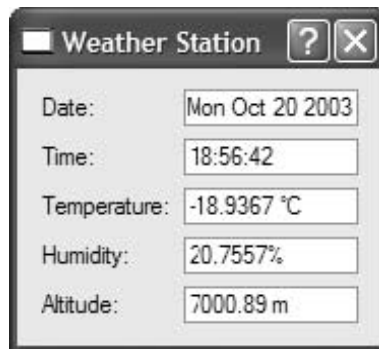



그림 13-3. Weather Station응용프로그람

Weather Balloon과 Weather Station실례를 통하여 Qt응용프로그람으로부터 UDP의 사용법을 알게 된다. Weather Balloon응용프로그람은 5s간격으로 현재 대기조건을 포함하는 UDP데이터그램을 보내는 비GUI응용프로그람이다. Weather Station응용프로그람은 이 데이터그램들을 수신하여 화면에 현시한다. Weather Balloon의 코드를 고찰하는것부터 시작한다.

```
class WeatherBalloon : public QPushButton
{
```

```
    Q_OBJECT
public:
    WeatherBalloon(QWidget *parent = 0, const char *name = 0);
    double temperature() const;
    double humidity() const;
    double altitude() const;
protected:
    void timerEvent(QTimerEvent *event);
private:
    QSocketDevice socketDevice;int myTimerId;
};
```

WeatherBalloon클라스는 QPushButton을 계승한다. 이것은 Weather Station과 교제하는데 그 QSocketDevice비공개변수를 사용한다.

```
WeatherBalloon::WeatherBalloon(QWidget *parent, const char *name)
    : QPushButton(tr("Quit"), parent, name), socketDevice(QSocketDevice::Datagram)
{
    socketDevice.setBlocking(false);
    myTimerId = startTimer(5 * 1000);
}
```

구성자의 초기화목록에서는 QSocketDevice::Datagram을 QSocketDevice구성자에 넘기여 UDP소케트장치를 창조한다. 구성자본체에서는 setBlocking(false)를 호출하여 QSocketDevice를 비동기로 만든다. (기정으로 QSocketDevice는 동기적이다.)

startTimer()를 호출하여 5s간격으로 시계사건을 생성한다.

```
void WeatherBalloon::timerEvent(QTimerEvent *event)
{
        if (event->timerId() == myTimerId) {
            QByteArray datagram;
            QDataStream out (datagram, IO_WriteOnly);
            out.setVersion(5);
            out << QDateTime::currentDateTime() << temperature() << humidity() << altitude();
            socketDevice.writeBlock(datagram, datagram.size(), 0x7F000001, 5824);
        } else {
            QPushButton::timerEvent(event);
        }
}
```

시계사건처리함수에서는 현재 날자, 시간, 온도, 습도, 표고를 포함하는 데이터그램을 생성한다.

QDateTime	측정날자와 시간
double	온도(℃)
double	습도(%)
double	표고(m)

데이터그램은 writeBlock()에 의해 송신된다. writeBlock()의 셋째와 넷째 인수는 IP주소와 동료(Weather Station)의 포구번호이다. 이 실례에서는 Weather Balloon과 같은 콤퓨터에서 Weather Station를 실행하고있다고 가정하므로 국부주콤퓨터를 지정하는 특수IP주소인 127.0.0.1 (0x7F000001)을 사용한다. QSocket와 달리 QSocketDevice는 주콤퓨터이름을 받아들이지 않고 주콤퓨터번호만 받아들인다. 여기서 주콤퓨터이름을 IP주소로 해결하려고 한다면 QDns클라스를 사용해야 한다.

보통과 같이 main()함수가 필요하다.

```
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    WeatherBalloon balloon;
    balloon.setCaption(QObject::tr("Weather Balloon"));
    app.setMainWidget(&balloon);
    QObject::connect(&balloon, SIGNAL(clicked()), &app, SLOT(quit()));
    balloon.show();
    return app.exec();
}
```

main()함수는 단순히 WeatherBalloon객체를 창조하고 이 객체는 UDP동료로서 그리고 화면우의 QPushButton으로서 모두 작업한다. QPushButton을 찰칵하여 사용자는 응용프로그람을 중지할수 있다.

이제는 Weather Station의 원천코드를 고찰하자.

```
class WeatherStation : public QDialog
{
    Q_OBJECT
public:
    WeatherStation(QWidget *parent = 0, const char *name = 0);
private slots:
    void dataReceived();
```

```
private:
    QSocketDevice  socketDevice;
    QSocketNotifier  *socketNotifier;
    QLabel  *dateLabel;
    QLabel  *timeLabel;
    ...
    QLineEdit *altitudeLineEdit;
};
```

WeatherStation클라스는 QDialog를 계승한다. 이 클라스는 일정한 UDP포구를 청취하여 Weather Balloon으로부터 들어오는 데이터그램들을 해석하고 5개의 읽기전용QLineEdit들에 그 내용을 현시한다.

클라스에는 2개의 비공개변수 socketDevice와 socketNotifier가 있다. socketDevice변수는 QSocketDevice형으로서 데이터그램들을 읽어들이는데 쓰인다. socketNotifier변수는 QSocketNotifier형이고 응용프로그람이 들어오는 데이터그램들을 인식하게 만든다.

```
WeatherStation::WeatherStation(QWidget *parent, const char *name)
     : QDialog(parent, name), socketDevice (QSocketDevice::Datagram)
{
    socketDevice.setBlocking(false);
    socketDevice.bind(QHostAddress(), 5824);
    socketNotifier = new QSocketNotifier(socketDevice.socket(), QSocketNotifier::Read, this);
    connect(socketNotifier, SIGNAL(activated(int)), this, SLOT(dataReceived()));
    ...
}
```

구성자의 초기화목록에서는 QSocketDevice::Datagram을 QSocketDevice구성자에 넘기여 UDP소케트장치를 창조한다. 구성자본체에서는 setBlocking(false)를 호출하여 소케트를 비동기로 만들고 bind()를 호출하여 포구번호를 소케트에 할당한다. 첫째 인수는 Weather Station의 IP주소이다. QHostAddress()를 넘김으로써 Weather Station가 실행중에 있는 콤퓨터에 속하는 IP주소에 데이터그램들을 받아들인다는것을 지적한다. 둘째 인수는 포구번호이다.

그다음 소케트를 감시하는 QSocketNotifier객체를 창조한다. 소케트가 데이터그램을 수신할 때마다 QSocketNotifier는 activated(int)신호를 발생한다. 그 신호를 dataReceived()처리부에 련결한다.

```
void WeatherStation::dataReceived()
{
    QDateTime  dateTime;
    double  temperature;
```

```
        double  humidity;
        double  altitude;
        QByteArray  datagram(socketDevice.bytesAvailable());
        socketDevice.readBlock(datagram.data(),  datagram.size());
        QDataStream  in(datagram,  IO_ReadOnly);
        in.setVersion(5);
        in >> dateTime >> temperature >> humidity >> altitude;
        dateLineEdit->setText(dateTime.date().toString());
        timeLineEdit->setText(dateTime.time().toString());
        temperatureLineEdit->setText(tr("%1  °C").arg(temperature));
        humidityLineEdit->setText(tr("%1%").arg(humidity));
        altitudeLineEdit->setText(tr("%1  m").arg(altitude));
    }
```

dataReceived()에서는 QSocketDevice에 대하여 readBlock()를 호출하여 데이터그램을 읽어들인다. QByteArray::data()는 readBlock()가 채워넣는 QByteArray자료의 지적자를 돌려준다. 그다음 QDataStream를 리용하여 각이한 마당들을 얻고 사용자대면부를 갱신하여 수신한 정보를 표시한다. 응용프로그람의 견지에서 데이터그램들은 늘 자료의 한단위로 송수신된다. 이것은 임의의 바이트들이 유효하다면 꼭 하나의 데이터그램이 도착하였고 읽어들일수 있다는것을 의미한다.

```
    int main(int argc, char *argv[])
    {
        QApplication app(argc, argv);
        WeatherStation  station;
        app.setMainWidget(&station);
        station.show();
        return app.exec();
    }
```

끝으로 main()에서는 WeatherStation을 창조하고 그것을 응용프로그람의 기본창문부품으로 만든다.

이로서 UDP송신기와 수신기를 끝낸다. 응용프로그람들은 아주 간단하며 데이터그램들을 보내는 Weather Balloon과 그것들을 받아들이는 Weather Station이다. 현실세계의 대다수 응용프로그람들은 두 응용프로그람이 자기 소케트에 대한 읽기와 쓰기를 모두 요구한다. QSocketDevice클라스는 봉사기에서 어느 주소와 포구에 응답하는가를 결정하는데 사용하는 peerAddress()와 peerPort()함수를 가지고있다.

제14장. XML

XML(Extensible Markup Language)은 자료교환과 자료보관에서 큰 인기를 끌고있는 본문파일형식이다.

Qt는 XML문서를 처리하기 위한 2가지 다른 API를 제공한다.

· SAX(Simple API for XML)는 가상함수들을 통하여 응용프로그람에 직접 사건해석을 통보한다.

· DOM(Document Object Model)은 XML문서를 응용프로그람이 항행할수 있는 나무구조로 변환한다.

특정한 응용프로그람에서 DOM과 SAX중 하나를 선택할 때 고려해야 할 인자들이 많다. SAX는 준위가 낮고 보통 빠르다. 이것은 단순한 과제들(XML문서에서 주어진 꼬리표의 출현을 모두 찾는것 등)과 기억기를 많이 소비하는 대규모파일들을 읽어들이는데 특히 적합하다. 그러나 대부분의 응용프로그람들에서 DOM이 제공하는 편리성은 SAX의 잠재적인 속도와 기억기리득보다 더 크다.

이 장에서는 두 API를 리용하여 XML파일들을 읽는 방법과 XML파일들을 작성하는 방법을 설명한다. 이 장은 XML의 기초지식을 전제로 하고있다.

제1절. SAX에 의한 XML읽기

SAX는 사실상 XML문서들을 읽어들이기 위한 공개적인 표준Java API이다. Qt의 SAX클라스들은 SAX2 Java실현후에 모형화되고 Qt관례에 일치시켰으므로 좀 차이가 있다.

Qt는 QXmlSimpleReader라고 부르는 SAX에 기초하는 비합법적인 XML문장해석기를 제공한다. 이 문장해석기는 잘 형식화된 XML을 인식하고 XML이름공간들을 유지한다. 문장해석기가 문서를 읽어들일 때 등록된 처리함수클라스들의 가상함수들을 호출하여 문장해석사건들을 가리킨다. (이 《문장해석사건》들은 건과 마우스사건들과 같은 Qt사건들에 관련되지 않는다.) 례를 들면 문장해석기가 다음의 XML문서를 해석하고있다고 가정한다.

```
<doc>
    <quote>Errare humanum est</quote>
</doc>
```

문장해석기는 다음과 같은 문장해석사건처리함수들을 호출한다.

```
startDocument()
startElement("doc")
startElement("quote")
characters("Errare humanum est")
endElement("quote")
endElement("doc")
```

endDocument()

우의 함수들은 모두 QXmlContentHandler에서 선언된다. 단순성을 위하여 startElement()와 endElement()의 일부 인수들을 생략하였다.

QXmlContentHandler는 QXmlSimpleReader와 결합하여 사용할수 있는 수많은 처리함수클라스들중 하나이다. 그밖에 QXmlEntityResolver, QXmlDTDHandler, QXmlErrorHandler, QXmlDeclHandler, QXmlLexicalHandler가 있다. 이러한 클라스들은 오직 순수가상함수들만 선언하며 각종 문장해석사건들에 대한 정보를 준다. 대부분의 응용프로그람들에서는 오직 QXmlContentHandler와 QXmlErrorHandler 두개만 요구한다.

또한 편리상 Qt는 다중계승을 통하여 모든 처리함수클라스들을 계승하며 모든 함수들의 일반적실현을 주는 QXmlDefaultHandler클라스를 제공한다. 이 설계는 많은 추상처리함수클라스들과 하나의 일반파생클라스를 가지는데 이것은 일반적으로 Qt가 아니라 Java실현모형에 기초하고있다.

이제는 QXmlSimpleReader와 QXmlDefaultHandler를 사용하여 특별한 XML파일형식을 해석하고 그 리용을 QListView에 표시하는 방법을 보여주는 실례를 고찰한다. QXmlDefaultHandler 파생클라스를 SaxHandler라고 하고 그 처리형식은 색인항목과 보조항목들을 가지는 도서색인이다.

QXmlContentHandler QXmlDTDHandler QXmlLexicalHandler
QXmlErrorHandler QXmlEntityResolver QXmlDeclHandler
QXmlDefaultHandler
SaxHandler

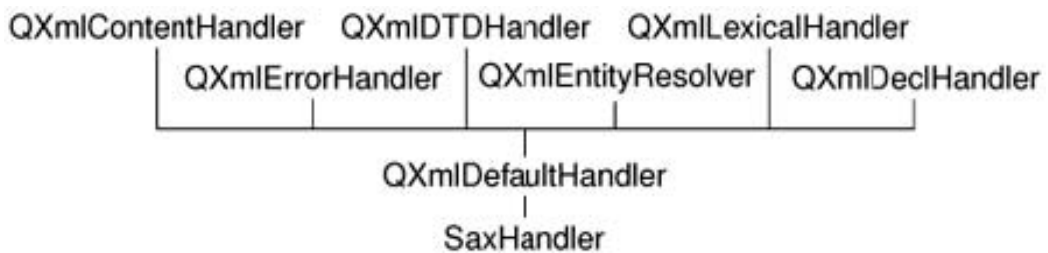



그림 14-1. SaxHandler의 계승나무

여기에 그림 14-2의 QListView에 현시되는 도서색인파일이 있다.

Terms	Pages
sidebearings	10, 34-35, 307-308
subtraction	
of pictures	115, 244
of vectors	0

그림 14-2. QListView에 적재된 도서색인파일

```
<?xml version="1.0"?>
<bookindex>
    <entry term="sidebearings">
        <page>10</page>
        <page>34-35</page>
```

```
            <page>307-308</page>
        </entry>
        <entry term="subtraction">
            <page>115</page>
            <page>244</page>
        </entry>
        <entry term="of pictures">
            <page>115</page>
            <page>244</page>
        </entry>
        <entry term="of vectors">
            <page>9</page>
        </entry>
    </entry>
</bookindex>
```

문장해석기를 실현하는 첫 단계는 QXmlDefaultHandler의 파생클라스를 만드는것이다.

```
class SaxHandler : public QXmlDefaultHandler
{
public:
    SaxHandler(QListView *view);
    bool startElement(const QString &namespaceURI, const QString &localName,
                      const QString &qName, const QXmlAttributes &attribs);
    bool endElement(const QString &namespaceURI,
    const QString &localName,
    const QString &qName);
    bool characters(const QString &str);bool fatalError(const QXmlParseException &exception);
private:
    QListView *listView;
    QListViewItem *currentItem;
    QString currentText;
};
```

Sax처리함수클라스는 QXmlDefaultHandler를 계승하며 4개의 함수 startElement(), endElement(), characters(), fatalError()를 재정의한다. 처음 3개 함수는 QXmlContentHandler에서 선언되고 마지막 함수는 QXmlErrorHandler에서 선언된다.

```
SaxHandler::SaxHandler(QListView *view)
{
    listView = view;
    currentItem = 0;
```

```
}
```

SaxHandler구성자는 XML파일에 보관된 정보를 현시하려는 QListView를 받아들인다.

```
bool SaxHandler::startElement(const QString &, const QString &, const QString &qName,
        const QXmlAttributes &attribs)
{
    if (qName == "entry") {
        if (currentItem) {
            currentItem = new QListViewItem(currentItem);
        } else {
            currentItem = new QListViewItem(listView);
        }
        currentItem->setOpen(true);
        currentItem->setText(0, attribs.value("term"));
    } else if (qName == "page") {
        currentText = "";
    }
    return true;
}
```

startElement()함수는 읽기프로그람이 새로운 열기꼬리표를 만날 때 호출된다. 셋째 파라메터는 꼬리표이름(혹은 더 정확하게는 그 《수식이름(qualified name)》)이다. 넷째 파라메터는 속성목록이다. 이 실례에서는 첫째와 둘째 파라메터들을 무시한다. 그것들은 XML의 이름공간기구를 사용하는 XML파일에 사용할수 있다.

꼬리표가 <entry>이면 새로운 QListView항목을 창조한다. 꼬리표가 다른 <entry>꼬리표안에 겹쌓이면 새 꼬리표는 색인안의 보조항목을 정의하고 둘러싸는 항목을 표시하는 QListViewItem의 자식으로서 QListViewItem을 창조한다. 그렇지 않으면 부모로서 listView를 가지는 QListViewItem을 창조하여 그것을 제일 웃준위항목으로 만든다. 항목에 대하여 setOpen(true)를 호출하여 그 자식들을 표시하고 setText()를 호출하여 렬 0에 표시된 본문을 <entry>꼬리표의 항속성의 값으로 설정한다.

꼬리표가 <page>이면 currentText를 빈 문자렬로 설정한다. currentText는 <page>와 </page> 꼬리표들사이에 배치된 본문을 보관한다.

끝으로 true를 돌려주어 SAX에게 파일해석을 계속하라고 알린다. 알려지지 않은 꼬리표들을 오유로서 통보하려면 false를 돌려줄수 있다. 또한 그때 QXmlDefaultHandler로부터 errorString()를 재정의하여 적당한 오유통보문을 돌려준다.

```
bool SaxHandler::characters(const QString &str)
{
```

```
    currentText += str;
    return true;
}
```

characters()함수는 XML문서의 문자자료를 알리기 위하여 호출된다. 간단히 문자들을 currentText변수에 추가한다.

```
bool SaxHandler::endElement(const QString &, const QString &, const QString &qName)
{
    if (qName == "entry") {
        currentItem = currentItem->parent();
    } else if (qName == "page") {
        if (currentItem) {
            QString allPages = currentItem->text(1);
            if (!allPages.isEmpty())
                allPages += ", ";
            allPages += currentText;
            currentItem->setText(1, allPages);
        }
    }
    return true;
}
```

endElement()함수는 읽기프로그람이 닫기꼬리표를 만날 때 호출된다. startElement()에서와 같이 셋째 파라메터는 꼬리표이름이다.

꼬리표가 </entry>이면 currentItem비공개변수를 갱신하여 현재 QListViewItem의 부모를 가리킨다. 이것은 currentItem변수가 대응하는 </entry>꼬리표를 읽기전에 보관한 값을 되살리도록 한다.

꼬리표가 </page>이면 지정된 폐지번호를 추가하거나 1렬 현재항목의 본문에서 반점으로 구분된 목록까지의 폐지범위를 추가한다.

```
bool SaxHandler::fatalError(const QXmlParseException &exception)
{
    qWarning("Line %d, column %d: %s", exception.lineNumber(), exception.columnNumber(),
            exception.message().ascii());
    return false;
}
```

fatalError()함수는 읽기프로그람이 XML파일의 문장해석에서 실패할 때 호출된다. 이런 일이 생기면 단순히 행번호, 렬번호, 문장해석기의 오유를 주는 경고를 출력한다.

이로서 Sax처리함수클라스의 실현을 끝낸다. 그러면 클라스의 사용방법을 고찰하자.

```
bool parseFile(const QString &fileName)
{
    QListView *listView = new QListView(0);
    listView->setCaption(QObject::tr("SAX Handler"));
    listView->setRootIsDecorated(true);
    listView->setResizeMode(QListView::AllColumns);
    listView->addColumn(QObject::tr("Terms"));
    listView->addColumn(QObject::tr("Pages"));
    listView->show();
    QFile file(fileName);
    QXmlSimpleReader reader;
    SaxHandler handler(listView);
    reader.setContentHandler(&handler);
    reader.setErrorHandler(&handler);
    return reader.parse(&file);
}
```

2개 렬을 가지도록 QListView을 설정한다. 그다음 읽으려는 파일용으로 QFile객체를 창조하고 파일을 해석하려는 QXmlSimpleReader를 창조한다. QFile을 자체로 열 필요는 없고 Qt가 자동적으로 연다.

끝으로 SaxHandler객체를 창조하고 그것을 읽기프로그람에 대하여 내용처리함수와 오유처리함수의 두 처리함수로 설치하고 읽기프로그람에 대하여 parse()를 호출하여 문장해석을 수행한다.

SaxHandler에서는 오직 QXmlContentHandler와 QXmlError처리함수클라스로부터 함수들을 재정의한다. 다른 처리함수클라스로부터 함수들을 실현하였다면 읽기기구에 대하여 대응하는 설정함수들을 호출해야 한다.

제2절. DOM에 의한 XML읽기

DOM은 W3C(World Wide Web Consortium)에 의해 개발된 XML해석용 표준API이다. Qt는 XML문서들을 읽고 조작하고 쓰기 위한 비합법적인 DOM준위2실현을 제공한다.

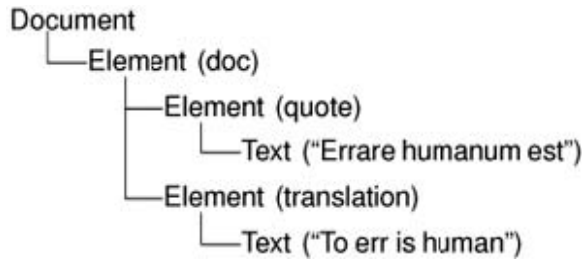
DOM은 XML파일을 기억기에 나무로 표시한다. 필요할 때 DOM나무를 항행할수 있고 나무를 수정하여 디스크에 XML파일로 보관할수 있다.

다음과 같은 XML문서를 고찰하자.

```
<doc>
    <quote>Errare humanum est</quote>
    <translation>To err is human</translation>
```

```
</doc>
```

이것은 다음의 DOM나무에 대응한다.

```
Document
 └─Element (doc)
     ├─Element (quote)
     │  └─Text ("Errare humanum est")
     └─Element (translation)
        └─Text ("To err is human")
```

DOM나무는 각이한 형의 마디들을 포함한다. 례를 들면 Element마디는 열기꼬리표와 그와 짝을 이루는 닫기꼬리표에 대응한다. 꼬리표들사이에 놓이는 자료는 Element마디의 자식마디들로 표시된다.

Qt에서 마디형들은 다른 모든 DOM관련클라스들처럼 QDom앞붙이를 가진다. 이리하여 QDomElement는 Element마디를, QDomText는 Text마디를 각각 표시한다.

각이한 형의 마디들은 각종 자식마디를 가질수 있다. 례를 들면 Element마디는 다른 Element마디들과 또한 EntityReference, Text, CDATASection, ProcessingInstruction 그리고 Comment 마디들을 포함할수 있다. 그림 14-3은 어느 마디들이 어떤 종류의 자식마디들을 가지는가 지적한다. 재색으로 표시된 마디들은 자체의 자식마디를 가질수 없다.

Document
Element
Document Type
Processing Instruction
Comment
Attr
Entity Reference
Text
Document Fragment
Element
Entity Reference
Entity
Element
Entity Reference
Text
CDATA Section
Processing Instruction
Comment

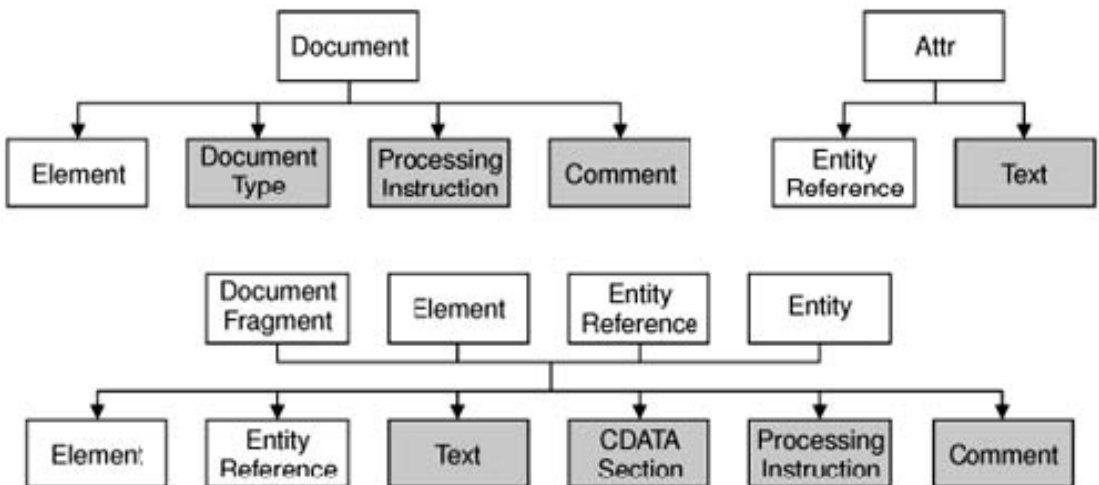



그림 14-3. DOM마디들사이의 부자관계

DOM을 리용하여 XML파일을 읽는 방법을 설명하기 위하여 앞 절에서 서술한 도서색인파일형식용의 문장해석기를 쓴다.

```
class  DomParser
{
public:
    DomParser(QIODevice  *device,  QListView  *view);
```

```
private:
    void parseEntry(const QDomElement &element, QListViewItem *parent);
    QListView *listView;
};
```

도서색인XML문서를 해석하는 DomParser라는 클래스를 정의하고 QListView에 결과를 현시한다. 이 클래스는 다른 클래스를 계승하지 않는다.

```
DomParser::DomParser(QIODevice *device, QListView *view)
{
    listView = view;
    QString errorStr;
    int errorLine;
    int errorColumn;
    QDomDocument doc;
    if (!doc.setContent(device, true, &errorStr, &errorLine, &errorColumn)) {
        qWarning("Line %d, column %d: %s", errorLine, errorColumn, errorStr.ascii());
        return;
    }
    QDomElement root = doc.documentElement();
    if (root.tagName() != "bookindex") {
        qWarning("The file is not a bookindex file");
        return;
    }
    QDomNode node = root.firstChild();
    while (!node.isNull()) {
        if (node.toElement().tagName() == "entry")
            parseEntry(node.toElement(), 0);
        node = node.nextSibling();
    }
}
```

구성자에서는 QDomDocument객체를 창조하고 그것에 대하여 setContent()를 호출하여 QIODevice가 제공하는 XML문서를 읽어들인다. setContent()함수는 장치가 이미 열려져 있지 않으면 그것을 자동적으로 연다. 그다음 QDomDocument에 대하여 documentElement()를 호출하여 하나의 QDomElement자식을 얻고 그것이 <bookindex>요소인가 검사한다. 그다음 모든 자식마디들을 순환하면서 마디가 <entry>요소이면 parseEntry()를 호출하여 해석한다.

QDomNode클라스는 임의의 형의 마디를 보관할수 있다. 마디를 더 처리하려고 한다면 우

선 마디를 정확한 자료형으로 변환해야 한다. 이 실례에서는 오직 Element마디들만 고찰하므로 QDomNode에 대하여 toElement()를 호출하여 QDomElement로 변환한 다음 tagName()을 호출하여 요소의 꼬리표이름을 얻는다. 마디가 Element형이 아니면 toElement()함수는 빈 꼬리표이름을 가지는 빈 QDomElement객체를 돌려준다.

```
void DomParser::parseEntry(const QDomElement &element, QListViewItem *parent)
{
    QListViewItem *item;
    if (parent) {
        item = new QListViewItem(parent);
    } else {
        item = new QListViewItem(listView);
    }
    item->setOpen(true);
    item->setText(0, element.attribute("term"));
    QDomNode node = element.firstChild();
    while (!node.isNull()) {
        if (node.toElement().tagName() == "entry") {
            parseEntry(node.toElement(), item);
        } else if (node.toElement().tagName() == "page") {
            QDomNode childNode = node.firstChild();
            while (!childNode.isNull()) {
                if (childNode.nodeType() == QDomNode::TextNode) {
                    QString page = childNode.toText().data();
                    QString allPages = item->text(1);
                    if (!allPages.isEmpty())
                        allPages += ", ";
                    allPages += page;
                    item->setText(1, allPages);
                    break;
                }
                childNode = childNode.nextSibling();
            }
        }
        node = node.nextSibling();
    }
```

}

parseEntry()에서는 QListView항목을 창조한다. 끄리표가 다른 <entry>끄리표안에 겹쌓이면 새 끄리표는 색인안의 보조항목을 정의하고 둘러싸고있는 항목을 표시하는 QListViewItem의 자식으로서 QListViewItem을 창조한다. 그렇지 않으면 부모로서 listView를 가지는 QListViewItem을 창조하여 그것을 제일 웃준위항목으로 만든다. 그 항목에 대하여 setOpen(true)를 호출하여 보조항목을 볼수 있도록 하며 setText()를 호출하여 렬 0에 표시된 본문을 <entry>끄리표의 term속성값으로 설정한다.

QListViewItem을 초기화한 다음 현재 <entry>끄리표에 대응하는 QDomElement마디의 자식마디들을 순환한다.

요소가 <entry>이면 현재 항목을 둘째 인수로 하여 parseEntry()를 호출한다. 그때 새 항목의 QListViewItem은 둘러싸인 항목의 QListViewItem을 부모로 하여 창조된다.

요소가 <page>이면 요소의 자식목록을 항행하여 Text마디를 찾는다. 그것을 발견하였으면 toText()를 호출하여 QDomText객체로 변환하고 data()를 호출하여 본문을 QString으로서 꺼낸다. 그다음 그 본문을 QListViewItem의 렬1안에서 반점으로 구분한 목록에 추가한다.

그러면 DomParser클라스에 의하여 파일을 해석하는 방법을 고찰하자.

```
void parseFile(const QString &fileName)
{
    QListView *listView = new QListView(0);
    listView->setCaption(QObject::tr("DOM Parser"));
    listView->setRootIsDecorated(true);
    listView->setResizeMode(QListView::AllColumns);
    listView->addColumn(QObject::tr("Terms"));
    listView->addColumn(QObject::tr("Pages"));
    listView->show();
    QFile file(fileName);
    DomParser(&file, listView);
}
```

QListView의 설정으로 시작한다. 그다음 QFile과 DomParser를 창조한다. DomParser가 구성될 때 파일을 해석하고 목록보기를 채운다.

실례에서 설명하는것처럼 DOM나무항행은 귀찮을수 있다. <page>와 </page>사이의 본문을 간단히 발취하려면 firstChild()와 nextSibling()을 리용하여 QDomNodes목록을 순환할것을 요구한다. DOM을 많이 사용하는 프로그람작성자들은 끄리표들사이의 본문을 꺼내는것과 같은 일반적으로 필요한 조작을 단순화하기 위하여 자체의 고수준래퍼함수들을 작성한다.

제3절. XML쓰기

Qt응용프로그람들로부터 XML파일들을 생성하는데 기본적으로 2가지 수법이 있다.

· DOM나무를 구축하고 그것에 대하여 save()를 호출한다.

· 코드를 작성하여 XML을 생성할수 있다.

이 수법들사이의 선택은 흔히 XML문서들을 읽어들이는데 SAX를 사용하는가 DOM을 사용하는가에 의존한다.

여기에 DOM나무를 창조하고 QTextStream에 의하여 그것을 써넣는 방법을 설명하는 코드부분이 있다.

```
const int Indent = 4;
QDomDocument doc;
QDomElement root = doc.createElement("doc");
QDomElement quote = doc.createElement("quote");
QDomElement translation = doc.createElement("translation");
QDomText quoteText = doc.createTextNode("Errare humanum est");
QDomText translationText = doc.createTextNode("To err is human");
doc.appendChild(root);
root.appendChild(quote);
root.appendChild(translation);
quote.appendChild(quoteText);
translation.appendChild(translationText);
QTextStream out(&file);
doc.save(out, Indent);
```

save()의 둘째 인수는 사용하려는 들여쓰기를 나타낸다. 들여쓰기의 크기는 0아닌 값으로 설정하면 사람들이 파일을 읽기 쉽다. 여기에 XML파일출력이 있다.

```
<doc>
    <quote>Errare humanum est</quote>
    <translation>To err is human</translation>
</doc>
```

다른 대본은 DOM나무를 기본자료구조로 사용하는 응용프로그람들에 있다. 이러한 응용프로그람들은 보통 DOM을 리용하여 XML문서들을 읽어들인 다음 기억기에서 DOM나무를 수정하고 끝으로 save()를 호출하여 나무를 XML로 변환한다.

우의 실례에서는 부호화로서 UTF-8을 사용한다. DOM나무에

```
<?xml version="1.0" encoding="ISO-8859-1"?>
```

를 종속시킴으로써 다른 부호화를 사용할수 있다. 다음의 코드부분은 이것을 수행하는 방법을 보여준다.

```
QTextStream out(&file);
QDomNode xmlNode =
```

```
        doc.createProcessingInstruction("xml","version=\"1.0\"  encoding=\"ISO-8859-1\"");
    doc.insertBefore(xmlNode,  doc.firstChild());
    doc.save(out, Indent);
```

수동적으로 XML파일들을 생성하여 DOM을 리용하는것보다 힘들지 않다. QTextStream을 리용하여 문자렬들을 임의의 본문파일에 대하여 수행한것처럼 써넣을수 있다. 가장 엄격한 부분은 본문과 속성값들에서 특수문자들을 확장하는것이다. 개별적인 함수에서 이것을 수행할수 있다.

```
QString  escapeXml(const  QString  &str)
{
    QString  xml  =  str;
    xml.replace("&",  "&amp;");
    xml.replace("<",  "&lt;");
    xml.replace(">",  "&gt;");
    xml.replace("'",  "&apos;");
    xml.replace("\"",  "&quot;");
    return xml;
}
```

여기에 그것을 사용하는 실례가 있다.

```
QTextStream  out(&file);
out.setEncoding(QTextStream::UnicodeUTF8);
out  <<  "<doc>\n"  <<  "  <quote>"  <<  escapeXml(quoteText)  <<  "</quote>\n"
      <<  "  <translation>"  <<  escapeXml(translationText)  <<  "</translation>\n"  <<  "</doc>\n"
```

제15장. 국제화

이 장에서는 영어가 아닌 언어로 Qt응용프로그람들을 쓰는 방법과 현존Qt응용프로그람을 다른 언어로 번역하는 방법을 설명한다.

1절에서는 Qt의 원시문자부호화인 유니코드를 론한다. 영어사용자대면부를 가지는 응용프로그람도 어느 날에는 우리글이나 그리스문으로 된 사용자의 콤퓨터에서 실행할수 있으므로 이 절에서 나오는 내용은 모든 Qt개발자들이 알아야 한다.

2절에서는 응용프로그람들의 번역준비방법을 보여준다. 이 과정은 너무 간단하므로 자기 쏘프트웨어의 번역판을 제공할 계획을 가지지 않는 경우에도 수행할만한 가치가 있다. 이것은 번역기를 빌리여 후날에 자기 응용프로그람용의 새로운 시장을 창조할 좋은 기회를 준다.

3절은 진짜 국제적인 응용프로그람들을 목적하여 실행중에 응용프로그람이 언어를 변경하게 하는 방법을 보여준다.

마지막 절은 번역과정을 모두 서술한다. 또한 Qt Linguist와 Qt의 다른 번역도구들을 리용하여 프로그람작성자와 번역기들이 함께 작업하는 방법을 보여준다.

제1절. 유니코드와의 작업

유니코드는 세계의 대다수 문서체계를 유지하는 문자부호화표준이다. 유니코드에 은페되여있는 원래의 사상은 문자보관에 8bit대신 16bit를 사용하여 256문자대신에 65000개의 문자들을 부호화하는것이였다. 유니코드는 ASCII와 ISO 8859-1(Latin-1)을 같은 코드위치에서 부분모임으로 포함한다. 례를 들면 문자 'A'는 ASCII와 Latin-1, 유니코드에서 값 0x41을 가지고 문자 'b'는 Latin-1과 유니코드량쪽에서 값 0xDF을 가진다.

Qt의 QString클라스는 문자렬들을 유니코드로 보관한다. QString안의 매개 문자는 8bit char가 아니라 16bit QChar이다. 여기에 문자렬의 첫 문자를 'A'로 설정하는 2가지 방법이 있다.

```
str[0] = 'A';
str[0] = QChar(0x41);
```

원천파일이 Latin-1로 부호화되면 Latin-1문자지정은 아주 간단하다.

```
str[0] = 'b';
```

그리고 원천파일이 다른 부호화를 가지면 수값은 제대로 작업한다.

```
str[0] = QChar(0xDF);
```

임의의 유니코드문자를 그 수값으로 지정할수 있다. 례를 들면 여기에 그리스대문자 시그마('Σ')와 유로화페기호('€')를 지정하는 방법이 있다.

```
str[0] = QChar(0x3A3);
str[0] = QChar(0x20AC);
```

드문히 Latin-1이 아닌 유니코드문자들이 요구되면 문자탐색은 직결로 충분히 할수 있다. 그러나 Qt는 Qt프로그람에 유니코드문자렬들을 입력하는 더 편리한 수법을 제공한다. 그것을

이 절에서 후에 설명한다.

Qt 3.2의 본문엔진은 모든 가동환경에서 다음과 같은 문서체계들을 유지한다. 아랍어, 중국어, 씨릴어, 그리스어, 헤브라이, 일본어, 조선어, 라오스어, 라틴어, 타이어, 및 웰남어. 또한 특수한 처리를 요구하지 않는 모든 유니코드 3.2스크립트들도 요구한다. 게다가 다음의 문서체계들은 Xft을 사용하는 X11과 NT기초의 Windows판들에 유지된다. 즉 벵갈어, Devanagari, Gujarati, Gurmukhi, Kannada, Khmer, 수리아어, 타밀어, Telugu, Thaana. 끝으로 Malayalam과 티베트어를 X11에 유지하고 Divehi를 Windows XP에 유지한다. 체계에 적당한 서체들이 설치된다고 가정하면 Qt는 이러한 문서체계중의 어느것이나 사용하여 본문을 그릴수 있다. 그리고 적당한 입력방법이 설치된다고 가정하면 사용자들은 Qt응용프로그람들에서 이러한 문서체계들을 사용하는 본문을 입력할수 있다.

QChar를 사용하는 프로그람작성은 char를 사용하는 프로그람작성과 아주 다르다. QChar의 수값을 얻으려면 그에 대하여 unicode()를 호출한다. QChar의 ASCII 혹은 Latin-1값(char)을 얻으려면 latin1()을 호출한다. Latin-1이 아닌 문자들에 대하여 latin1()은 0을 돌려준다.

프로그람안의 모든 문자렬들이 ASCII 혹은 Latin-1이라는것을 알고있으면 isalpha(), isdigit(), isspace()와 같은 표준<cctype>함수들을 사용할수 있다. 이것들은 QString들이 자동적으로 const char *로 변환되듯이 QChars가 정확한 문맥으로 주어진 char (Latin-1)들로 자동변환되므로 제대로 작업한다. 그러나 일반적으로 이러한 조작을 수행하는데 QChar의 성원함수들을 사용하는것이 더 좋다. 그것은 QChar의 성원함수들이 임의의 유니코드문자들에 대하여 작업하기때문이다. QChar가 제공하는 함수에는 isPrint(), isPunct(), isSpace(), isMark(), isLetter(), isNumber(), isLetterOrNumber(), isDigit(), isSymbol(), lower(), upper()가 있다. 례를 들면 여기에 문자가 수자인가 대문자인가를 시험하는 방법이 있다.

```
if (ch.isDigit() || ch != ch.lower()) ...
```

lower()함수는 소문자판의 문자를 돌려준다. 문자의 소문자판이 문자자체와 다르면 그 문자는 대문자이여야 한다. 코드부분은 라틴어, 그리스어, 시릴어를 비롯한 대소문자를 구별하는 임의의 자모에 대하여 작업한다.

QString을 요구하는 Qt의 API는 모두 유니코드문자렬을 사용할수 있다. 그때 유니코드문자렬을 적당히 현시하고 조작체계와 대화할 때 다른 부호화로 변환하는것이 Qt의 응답능력이다.

본문파일들을 읽고 쓸 때에는 특별한 주의가 필요하다. 본문파일들은 여러가지 부호화를 사용하며 그 문맥으로부터 본문파일의 부호화를 알아낼수는 없다. 기정으로 QTextStream은 읽기와 쓰기에 대하여 QTextCodec::codecForLocale()에서 유효한 체계의 국부8bit부호화를 사용한다. 미국과 서유럽지역에서 이것은 Latin-1을 의미한다.

자체의 파일형식을 설계하고 임의의 유니코드문자를 읽고 쓰게 하려면 QTextStream에 써넣기를 시작하기전에 setEncoding(QTextStream::Unicode)를 호출하여 자료를 유니코드로 보관할수 있다. 그때 자료는 문자당 2byte를 요구하는 형식인 UTF-16으로 보관된다. UTF-16형식은

QString의 기억기표시와 아주 근사하므로 유니코드문자렬들을 UTF-16으로 읽고쓰는 속도는 아주 빠르다. 그러나 순수 ASCII자료를 UTF-16형식으로 보관할 때 매개 문자를 1byte대신에 2byte로 보관하므로 추가비용이 든다.

본문을 읽어들일 때 QTextStream은 보통 유니코드를 자동적으로 탐지하지만 절대적인 확신을 위하여 읽기전에 setEncoding(QTextStream::Unicode)를 호출하는것이 제일 좋다.

유니코드전체를 유지하는 다른 부호화는 UTF-8이다. UTF16에 비한 UTF-8의 우점은 그것이 ASCII의 웃준위모임이라는것이다. 범위 0x00～0x7F의 임의의 문자는 1byte로 표시된다. 0x7F이상의 Latin-1문자들을 포함하는 다른 문자들은 여러바이트렬로 표시된다. 대체로 ASCII인 본문에서 UTF-8은 UTF-16이 소비하는 공간의 약 절반을 차지한다. UTF-8을 QTextStream에서 사용하려면 읽고쓰기전에 setEncoding(QTextStream::UnicodeUTF8)를 호출해야 한다.

사용자의 지역에 관계없이 늘 Latin-1을 읽고써넣으려고 한다면 QTextStream에 대하여 setEncoding(QTextStream::Latin1)을 호출할수 있다.

적당한 QTextCodec를 리용하여 setCodec()를 호출함으로써 다른 부호화를 지정할수 있다. QTextCodec는 유니코드와 주어진 부호화사이를 변환하는 객체이다. QTextCodec는 여러가지 문맥에서 Qt에 의해 사용된다. 내적으로 QTextCodec는 서체, 입력메쏘드, 오려둠판, 끌어다놓기, 파일이름들을 유지하는데 쓰인다. 그러나 Qt응용프로그람을 쓸 때에도 사용할수 있다.

례를 들면 EUC-KR부호화로서 파일을 읽어들이려고 한다면 다음과 같이 쓸수 있다.

```
QTextStream in(&file);
QTextCodec *koreanCodec = QTextCodec::codecForName("EUC-KR");
if (koreanCodec)
  in.setCodec(koreanCodec);
```

일부 파일형식들은 머리부에서 부호화를 지정한다. 일반적으로 머리부는 평본문ASCII로 함으로써 어떤 부호화를 사용하는가에 관계없이 정확히 읽어들이도록 한다. (그것이 ASCII의 웃준위모임이라고 가정한다.) XML파일형식은 그러한 실례의 하나이다. XML파일들은 보통 UTF-8이나 UTF-16로 부호화된다. 그 파일들을 읽어들이는 적당한 수법은 setEncoding(QTextStream::UnicodeUTF8)을 호출하는것이다. 형식이 UTF-16이면 QTextStream는 자동적으로 이것을 탐지하고 자체로 조절한다. XML파일의 <?xml?>머리부는 흔히 부호화인수를 포함한다. 례를 들면

```
<?xml version="1.0" encoding="EUC-KR"?>
```

일단 읽기 시작하면 QTextStream이 부호화를 변경하지 못하게 하므로 명시적인 부호화를 고려하는 옳은 방법은 정확한 부호(QTextCodec::codecForName()로부터 얻어진다.)를 사용하여 그 파일을 다시 읽기 시작하는것이다.

XML의 경우에는 14장에서 서술하는 Qt의 XML클라스들을 사용하여 자체로 부호화를 처리하는것이다.

또 하나의 QTextCodec사용은 원천코드에서 발생하는 문자렬들의 부호화를 지정하는것이

다. 일본의 가정용시장을 주로 목표로 삼은 응용프로그람들을 작성하는 일본어 프로그람작성자팀의 실례를 고찰하자. 이러한 프로그람작성자들은 EUC-JP 혹은 Shift-JIS와 같은 부호화를 사용하는 본문편집기에서 자기의 원천코드를 쓰고싶어한다. 그러한 편집기는 원천코드를 일본어로 원만히 입력하게 하므로 다음과 같이 코드를 쓸수 있다.

```
QPushButton *button = new QPushButton(tr("日諾"), 0);
```

기정으로 Qt는 tr()의 인수들을 Latin-1로 해석한다. 이것을 변경하려면 QTextCodec::setCodecForTr()정적함수를 호출해야 한다. 례를 들면

```
QTextCodec *japaneseCodec = QTextCodec::codecForName("EUC-JP");
QTextCodec::setCodecForTr(japaneseCodec);
```

이것은 처음으로 tr()를 호출하기전에 수행되여야 한다. 일반적으로 main()에서 QApplication객체를 창조한 직후에 수행할수 있다.

프로그람에서 지정된 다른 문자렬들은 여전히 Latin-1문자렬로 해석된다. 프로그람작성자들이 문자렬들에 일본어문자를 입력하려고 한다면 QTextCodec를 사용하여 문자들을 명시적으로 유니코드로 변환할수 있다.

```
QString text = japaneseCodec->toUnicode("海鮮料理");
```

또한 QTextCodec::setCodecForCStrings()를 호출하여 const char *와 QString사이를 변환할 때 특수부호를 사용하도록 Qt에 요구할수 있다.

```
QTextCodec::setCodecForCStrings(japaneseCodec);
```

Qt의 내부는 흔히 ASCII문자렬을 QString으로 변환하므로 부호화는 ASCII의 웃준위모임이여야 한다.

우에서 서술한 기술은 중국어, 그리스어, 조선어, 로어를 비롯하여 임의의 Latin-1이 아닌 언어에 적용할수 있다.

여기에 Qt 3.2가 유지하는 부호화의 목록이 있다.

Apple Roman	CP1258	ISO 8859-4	ISO 8859-15
Big5-HKSCS	EUC-JP	ISO 8859-5	ISO 10646
CP874	EUC-KR	ISO 8859-6	UCS-2
CP1250	GB2312	ISO 8859-7	JIS7
CP1251	GB18030	ISO 8859-8	KO18-R
CP1252	GBK	ISO 8859-8-I	KOI8-U
CP1253	IBM-850	ISO 8859-9	Shift-JIS
CP1254	IBM-866	ISO 8859-10	TIS-620
CP1255	ISO 8859-1	ISO 8859-11	TSCII
CP1256	ISO 8859-2	ISO 8859-13	UTF-8
CP1257	ISO 8859-3	ISO 8859-14	

이 모두에 대하여 QTextCodec::codecForName()는 늘 유효지적자를 돌려준다. QTextCodec의

파생클라스를 만들거나 문자략도(charmap)파일을 창조하고 QTextCodec::loadCharmapFile()을 사용함으로써 다른 부호화들을 유지할수 있다.(자세한 내용은 QTextCodec방조를 참고하시오.)

제2절. 번역을 인식하는 응용프로그람 만들기

다국어를 사용할수 있는 응용프로그람을 만들려고 한다면 다음의 2가지를 수행해야 한다.

· 사용자에게 보여주려는 문자렬이 tr()안에 들어있는가 확인한다.

· 기동시에 번역(.qm)파일을 적재한다.

번역하지 않는 응용프로그람들에는 이것이 필요없다. 그러나 tr()사용은 거의 품을 들이지 않고 후날에 번역을 진행하기 위한 문을 열수 있게 한다.

tr()함수는 QObject에서 정의된 정적함수이고 Q_OBJECT마크로로 정의된 모든 파생클라스에서 재정의된다. QObject파생클라스안에서 코드를 쓸 때에는 형식이 없이 tr()를 호출할수 있다. tr()호출은 번역이 유효이면 번역문을 돌려주고 그렇지 않으면 원래의 본문을 돌려준다.

번역파일들을 준비하려면 Qt의 lupdate도구를 실행하여야 한다. 이 도구는 tr()호출에 나타나는 문자렬상수들을 모두 꺼내서 이 문자렬들을 모두 포함하고 번역할 준비가 되여있는 번역파일을 생성한다. 그때 파일들은 번역기에 보내지고 추가된 번역문을 가지게 된다. 이 과정은 4절 《응용프로그람의 번역》에서 설명한다.

tr()호출은 다음과 같은 일반문법을 가진다.

Context::tr(*sourceText, comment*)

*Context*부분은 Q_OBJECT마크로에 의해 정의된 QObject의 파생클라스이름이다. 문제로 되여있는 클라스의 성원함수로부터 tr()를 호출한다면 그 마크로를 지정할 필요가 없다. *sourceText*부분은 번역해야 할 문자렬상수이다. *comment*부분은 선택적이고 번역기에 추가정보를 제공하는데 쓰인다.

여기에 몇가지 실례가 있다.

```
BlueWidget::BlueWidget(QWidget *parent, const char *name) : QWidget(parent, name)
{
    QString str1 = tr("Legal");
    QString str2 = BlueWidget::tr("Legal");
    QString str3 = YellowDialog::tr("Legal");
    QString str4 = YellowDialog::tr("Legal", "US paper size");
}
```

tr()의 첫 호출은 문맥으로서 BlueWidget를 가지며 마지막 2개의 호출은 YellowDialog를 가진다. 4개 모두가 Legal을 원천본문으로 가진다. 마지막 호출도 번역기가 원천본문의 의미를 리해하도록 방조하기 위한 설명문을 가진다.

각이한 문맥(클라스)의 문자렬들은 서로 독립적으로 번역된다. 보통 번역기들은 흔히 번역해야 할 창문부품이나 대화칸을 실행하고 표시하는 응용프로그람에서 한번에 하나의 문맥에 대하여 작업한다.

대역함수로부터 tr()를 호출할 때 문맥을 정확히 지정해야 한다. 응용프로그람안의 임의의 QObject파생클라스를 문맥으로 사용할수 있다. 적당한것이 없으면 늘 QObject자체를 사용할수 있다. 례를 들면

```
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    ...
    QPushButton button(QObject::tr("Hello Qt!"), 0);
    app.setMainWidget(&button);
    button.show();
    return app.exec();
}
```

이러한 수법을 응용프로그람의 이름을 번역하는데 사용할수 있다. 이름을 여러번 입력하고 번역기가 이름이 나타나는 매개 클라스에 대하여 그것을 번역하게 하는것이 아니라 번역된 응용프로그람이름으로 전개하는 APPNAME마크로를 정의하여 응용프로그람의 모든 파일에 포함되는 머리부파일에 마크로를 넣는것이 일반적으로 더 편리하다.

```
#define APPNAME MainWindow::tr("OpenDrawer 2D")
```

지금까지의 실례에서 문맥은 클라스이름이였다. 이것은 대체로 클라스이름을 생략할수 있으므로 편리하지만 늘 그런것은 아니다. Qt에서 문자렬을 번역하는 가장 일반적인 방법은 QApplications::translate()함수를 사용하는것이다. 이 함수는 3개 인수 즉 문맥, 원천본문, 설명문의 선택을 받아들인다. 례를 들면 여기에 APPNAME을 정의하는 다른 방법이 있다.

```
#define APPNAME qApp->translate("Global Stuff", "OpenDrawer 2D")
```

이번에는 본문을 "Global Stuff"문맥에 넣는다.

tr()와 translate()함수는 2중목적에 쓰인다. 즉 이 함수들은 lupdate가 사용자에게 보여주는 문자렬들을 찾는데 사용되는 표식기인 동시에 본문을 번역하는 C++ 함수들이다. 이것은 코드작성방법에 영향을 준다. 례를 들면 다음의 코드는 동작하지 않는다.

```
// WRONG
const char *appName = "OpenDrawer 2D";
QString translated = tr(appName);
```

여기서 문제는 "OpenDrawer 2D"문자렬상수가 tr()호출안에 나타나지 않으므로 lupdate가 꺼낼수 없는것이다. 이것은 번역기가 문자렬을 번역할 기회를 모른다는것을 의미한다. 이 문제는 흔히 동적문자렬과 관련하여 제기된다.

```
// WRONG statusBar()->message(tr("Host " + hostName + " found"));
```

여기서 tr()에 넘기는 문자렬은 hostName의 값에 의존하므로 tr()가 그것을 정확히 번역하리라고 기대할수 없다.

해결책은 QString::arg()를 사용하는것이다.

```
statusBar()->message(tr("Host %1 found") .arg(hostName));
```

그 동작을 알아두자. 문자렬상수 "Host %1 found"가 tr()에 넘어간다. 프랑스어번역파일을 적재하였다고 가정하면 tr()는 "Hôte %1 trouvé"와 같은것을 돌려준다. 그때 "%1" 파라메터는 hostName변수의 내용으로 교체된다.

일반적으로 변수에 대하여 tr()호출을 가능하게 하려면 QT_TR_NOOP()마크로를 사용하여 문자렬상수들을 변수에 대입하기전에 번역용으로 표식한다. 이것은 문자렬들의 정적배렬에 가장 쓸모있다. 례를 들면

```
void OrderForm::init()
{
    static const char * const flowers[] = {
       QT_TR_NOOP("Medium Stem Pink Roses"), QT_TR_NOOP("One Dozen Boxed Roses"),
       QT_TR_NOOP("Calypso Orchid"), QT_TR_NOOP("Dried Red Rose Bouquet"),
       QT_TR_NOOP("Mixed Peonies Bouquet"), 0
    };
    int i = 0;
    while (flowers[i]) {
       comboBox->insertItem(tr(flowers[i]));++i;}
   }
```

QT_TR_NOOP()는 단순히 그 인수를 돌려준다. 그러나 lupdate는 QT_TR_NOOP()에 포함되는 모든 문자렬들을 꺼내므로 문자렬들을 번역할수 있다. 후에 변수를 사용할 때 tr()를 호출하여 보통처럼 번역을 수행한다. 지어는 tr()에 변수를 넘기였다 할지라도 번역은 여전히 진행된다.

또한 QT_TR_NOOP()처럼 작업하지만 문맥을 가지는 QT_TRANSLATE_NOOP()마크로가 있다. 이 마크로는 클라스밖에서 변수들을 초기화할 때 쓸모있다.

```
static const char * const flowers[] = {
     QT_TRANSLATE_NOOP("OrderForm", "Medium Stem Pink Roses"),
     QT_TRANSLATE_NOOP("OrderForm", "One Dozen Boxed Roses"),
     QT_TRANSLATE_NOOP("OrderForm", "Calypso Orchid"),
     QT_TRANSLATE_NOOP("OrderForm", "Dried Red Rose Bouquet"),
     QT_TRANSLATE_NOOP("OrderForm", "Mixed Peonies Bouquet"),
};
```

context인수는 후에 tr() 혹은 translate()에 주어지는 문맥과 같아야 한다.

tr()를 응용프로그람에서 사용하기 시작할 때 사용자가 볼수 있는 문자렬들을 tr()호출안에 넣는것을 잊기 쉽다. 특히 처음 사용할 때는 더하다. tr()호출을 빠뜨린것들은 우연히 번역기가

혹은 번역된 응용프로그람의 사용자들이 발견하며 그때 일부 문자렬들은 원래 언어로 나타난다. 이 문제를 피하기 위하여 Qt에 const char *로부터 QString에로의 암시적변환을 금지하도록 한다. <qstring.h>를 포함하기전에 QT_NO_CAST_ASCII앞처리기기호를 정의함으로써 이것을 수행한다. 이 기호를 담보하는 가장 간단한 수법은 응용프로그람의 .pro파일에 다음행을 추가하는것이다.

```
DEFINES += QT_NO_CAST_ASCII
```

이것은 모든 문자렬상수가 번역되여야 하는가 아닌가에 따라 tr()나 QString::fromAscii()안에 넣을것을 요구한다. 적당히 넣어지지 않은 문자렬들은 콤파일시오유를 생성하므로 빠뜨린 tr() 혹은 QString::fromAscii()호출을 강제로 포함하게 한다.

일단 사용자에게 표시하는 모든 문자렬을 tr()호출안에 넣은 다음 번역이 가능하도록 하기 위하여 해야 할 유일한 일은 번역파일을 적재하는것이다. 일반적으로 응용프로그람의 main()함수에서 이것을 수행한다. 례를 들면 여기에 사용자의 지역에 따라 번역파일을 적재하는 코드가 있다.

```
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    QTranslator appTranslator;
    appTranslator.load(QString("app_") + QTextCodec::locale(), qApp->applicationDirPath());
    app.installTranslator(&appTranslator);
    ...
    return app.exec();
}
```

QTextCodec::locale()함수는 사용자의 지역을 지정하는 문자렬을 돌려준다. 지역은 정확하거나 그렇지 않을수 있다. 례를 들면 fr는 프랑스어지역을 지정하고 fr_CA는 프랑스어 카나다지역을 지정하고 fr_CA.ISO8859-15는 ISO 8859-15부호화('€', 'Œ', 'œ', 'Ÿ'를 유지하는 부호화)를 가지는 프랑스어 카나다지역을 지정한다.

지역이 fr_CA.ISO8859-15라고 가정하면 load()는 우선 파일 app_fr_CA.ISO8859-15.qm을 적재하려고 시도한다. 이 파일이 존재하지 않으면 load()는 다음으로 app_fr_CA.qm, 그다음 app_fr.qm, 끝으로 포기하기전에 app.qm를 적재하려고 한다. 보통 표준프랑스어번역을 포함하는 app_fr.qm만 제공하지만 프랑스어로 말하는 카나다용의 다른 파일을 요구한다면 app_fr_CA.qm도 제공할수 있으며 그것은 fr_CA지역들에 사용할수 있다.

load()의 둘째인수는 load()가 번역파일을 찾으려고 하는 등록부이다. 이 경우에는 번역파일들이 실행가능파일이 있는 등록부에 배치된다고 가정한다.

Qt서고자체는 번역할 필요가 있는 일부 문자렬들을 포함한다. Trolltech는 Qt의 translations 등록부에 프랑스어와 도이칠란드어번역을 제공한다. (일부 다른 언어들도 물론 제공되지만 이

것은 Qt사용자들이 설정하며 공식적으로 유지되지 않는다.) Qt서고의 번역파일도 적재되여야 한다.

```
QTranslator qtTranslator;qtTranslator.load(QString("qt_") + QTextCodec::locale(),
                    qApp->applicationDirPath());
app.installTranslator(&qtTranslator);
```

QTranslator객체는 한번에 오직 하나의 번역파일만 보유할수 있으므로 Qt번역마다 제각기 QTranslator를 사용한다. 번역기당 한개 파일을 주면 필요한만큼 번역기를 많이 설치할수 있으므로 문제시되지 않는다. QApplication은 번역을 탐색할 때 그것들을 모두 사용한다.

아랍어와 헤브라이어와 같은 일부 언어들은 왼쪽에서 오른쪽으로가 아니라 오른쪽에서 왼쪽으로 쓴다. 그러한 언어들에서 응용프로그람의 전체 배치는 역전되여야 하며 이것은 QApplication::setReverseLayout(true)을 호출하여 수행한다. Qt서고의 번역파일들은 언어가 왼쪽에서 오른쪽으로 쓰는가 오른쪽에서 왼쪽으로 쓰는가를 Qt에 알리는 "LTR"라는 특수한 표식기(marker)를 포함하므로 보통 그에 대하여 걱정하지 않아도 된다.

실행파일에 매몰된 번역파일들을 가지는 응용프로그람들을 공급한다면 사용자들에게 더욱 편리하다. 이것은 제품의 부분품으로 배포되는 파일수를 줄일뿐아니라 우연히 번역파일들을 잃어버리거나 삭제해버릴 위험성을 피하게 한다. Qt는 qembed도구(Qt의 tools등록부에 있다.)를 제공하는데 이 도구는 .qm파일들을 QTranslator::load()에 넘길수 있는 C++배렬로 변환한다.

이제는 다른 언어들에로의 번역을 사용하여 응용프로그람이 조작할수 있게 하는데 필요한 모든것에 대하여 설명하였다. 그러나 문서체계의 언어와 방향은 나라와 문화에 따라 달라지는 유일한 것이 아니다. 국제화된 프로그람은 또한 지역의 날자와 시간형식, 화페형식, 수자형식과 문자렬대조순서도 고려해야 한다. Qt 3.2는 이것들을 호출하는 고유한 함수들을 제공하지 않지만 표준C++의 setlocale()과 localeconv()함수를 사용하여 프로그람의 현재지역을 문의할수 있다.

일부 Qt클라스들과 함수들은 지역에 따르는 동작을 받아들인다.

· QString::localeAwareCompare()은 지역에 의존하는 방법으로 2개 문자렬을 비교한다.

이것은 QIconView와 QListView와 같은 클라스들에서 항목들을 정렬하는데 사용한다.

· QDate, QTime, QDateTime이 제공하는 toString()함수는 Qt::LocalDate를 인수로 하여 호출될 때 지역형식의 문자렬을 돌려준다.

· 기정으로 QDateEdit, QTimeEdit, QDateTimeEdit는 날자들을 지역형식으로 표시한다.

끝으로 번역된 응용프로그람은 원래의 그림기호가 아니라 상황에 따라 다른 그림기호들을 사용할것을 요구한다. 례를 들면 웨브열람기의 Back와 Forward단추상에서 왼쪽과 오른쪽화살표는 오른쪽에서 왼쪽으로 쓰는 언어를 취급할 때 교체되여야 한다. 이것을 다음과 같이 수행할수 있다.

```
if (QApplication::reverseLayout()) {
```

```
        backAct->setIconSet(forwardIcon);
        forwardAct->setIconSet(backIcon);
    } else {
        backAct->setIconSet(backIcon);
        forwardAct->setIconSet(forwardIcon);
    }
```

자모문자를 포함하는 그림기호들은 거의 모든 경우에 번역되여야 한다. 례를 들면 문서처리기의 도구띠단추에서 Italic선택과 련관된 문자 'I'는 에스빠냐어에서 'C' (*Cursivo*)로, 단마르크어, 네테를란드어, 도이췰란드어, 노르웨이어, 스웨리예어에서 'K'(*Kursiv*)로 교체되여야 한다. 여기에 그것을 수행하는 빠른 수법이 있다.

```
    if (tr("Italic")[0] == 'C') {
        italicAct->setIconSet(iconC);
    } else if (tr("Italic")[0] == 'K') {
        italicAct->setIconSet(iconK);
    } else {
        italicAct->setIconSet(iconI);
    }
```

제3절. 동적언어절환

대부분의 응용프로그람들에서는 main()에서 사용자의 언어를 탐지하여 적당한 .qm파일들을 적재하면 완전히 충족된다. 그러나 사용자들이 언어를 동적으로 절환할 필요가 있는 경우들이 있다. 여러 사람들이 번갈아 계속 사용하는 응용프로그람은 재기동하지 않고 언어를 바꿀것을 요구한다. 례를 들면 호출쎈터조종원, 동시통역원들, 그리고 콤퓨터화된 현금기록조종원들이 사용하는 응용프로그람들은 흔히 이러한 능력을 요구한다.

동적으로 언어를 절환할수 있는 응용프로그람을 만들려면 기동시에 하나의 번역을 적재하는 경우보다 좀 더 작업을 해야 하지만 어렵지 않다. 여기에 그 수행방법이 있다.

· 사용자가 언어를 절환할수 있는 수단들을 제공한다.

· 매개 창문부품이나 대화칸에서 번역할수 있는 문자렬들을 모두 개별적인 함수에 설정하고(흔히 retranslateStrings()를 호출하여) 언어가 달라질 때 이 함수를 호출한다.

Call Center응용프로그람의 원천코드에서 관련한 부분을 고찰하자. 응용프로그람은 Language차림표를 제공하여 사용자가 언어를 실행시에 설정하게 한다. 기정언어는 영어이다.

Language

العربية 1
2 Deutsch
3 English
4 Français
עברית 5
6 日本語

그림 15-1. Call Center응용프로그람의 Language차림표

응용프로그람이 기동할 때 사용자가 사용할 언어를 모르므로 main()함수에서 번역을 적재할 필요는 없다. 그대신에 필요할 때 번역을 동적으로 적재해야 하므로 번역을 처리하는 코드는 모두 기본창문과 대화칸클라스들에 넣어야 한다.

Call Center응용프로그람의 QMainWindow파생클라스를 고찰하자.

```
MainWindow::MainWindow(QWidget *parent, const char *name) : QMainWindow(parent, name)
{
    journalView = new JournalView(this);
    setCentralWidget(journalView);
    qmPath = qApp->applicationDirPath() + "/translations";
    appTranslator = new QTranslator(this);
    qtTranslator = new QTranslator(this);
    qApp->installTranslator(appTranslator);
    qApp->installTranslator(qtTranslator);
    createActions();
    createMenus();
    retranslateStrings();
}
```

구성자에서는 중심창문부품을 QListView의 파생클라스 JournalView로 설정한다. 그다음 번역과 관련한 여러개의 비공개성원변수들을 설정한다. 즉

· qmPath변수는 응용프로그람의 번역파일들을 포함하는 등록부의 경로를 지정하는 QString이다.

· appTranslator변수는 현재 응용프로그람번역을 보관하는데 사용된 QTranslator객체의 지적자이다.

· qtTranslator변수는 Qt의 번역을 보관하는데 사용되는 QTranslator객체의 지적자이다.

끝으로 createActions()와 createMenus()비공개함수들을 호출하여 차림표체계를 창조하고 비공개함수 retranslateStrings()를 호출하여 사용자에게 표시하는 문자렬들을 처음으로 설정한다.

```
void MainWindow::createActions()
{
```

```
    newAct = new QAction(this);
    connect(newAct, SIGNAL(activated()), this, SLOT(newFile()));
    ...
    aboutQtAct = new QAction(this);
    connect(aboutQtAct, SIGNAL(activated()), qApp, SLOT(aboutQt()));
}
```

createActions()함수는 보통처럼 QAction객체들을 창조하지만 본문이나 지름건들을 설정하지 않는다. 이것들은 retranslateStrings()에서 수행한다.

```
void MainWindow::createMenus()
{
    fileMenu = new QPopupMenu(this);
    newAct->addTo(fileMenu);
    openAct->addTo(fileMenu);
    saveAct->addTo(fileMenu);
    exitAct->addTo(fileMenu);
    ...
    createLanguageMenu();
}
```

createMenus()함수는 차림표들을 창조하지만 차림표띠에 삽입하지 않는다. 이것도 역시 retranslateStrings()에서 수행한다.

함수의 끝에서는 createLanguageMenu()를 호출하여 Language차림표를 유지된 언어들의 목록으로 채운다. 그 원천코드를 고찰한다. 우선 retranslateStrings()를 고찰한다.

```
void MainWindow::retranslateStrings()
{
    setCaption(tr("Call Center"));
    newAct->setMenuText(tr("&New"));
    newAct->setAccel(tr("Ctrl+N"));
    newAct->setStatusTip(tr("Create a new journal"));
    ...
    aboutQtAct->setMenuText(tr("About &Qt"));
    aboutQtAct->setStatusTip(tr("Show the Qt library's About box"));
    menuBar()->clear();
    menuBar()->insertItem(tr("&File"), fileMenu);
    menuBar()->insertItem(tr("&Edit"), editMenu);
    menuBar()->insertItem(tr("&Reports"), reportsMenu);
```

```
    menuBar()->insertItem(tr("&Language"), languageMenu);
    menuBar()->insertItem(tr("&Help"), helpMenu);
}
```

retranslateStrings()함수는 MainWindow클라스용의 모든 tr()호출이 발생하는 곳이다. 이 함수는 MainWindow구성자의 끝에서 호출되고 또한 사용자가 Language차림표에 의해 응용프로그람의 언어를 변경할 때마다 호출된다.

매개 QAction의 차림표본문, 지름건, 상태암시를 설정한다. 또한 차림표들을 번역된 이름으로 차림표띠에 삽입한다. (clear()호출은 retranslateStrings()를 한번이상 호출할 때 필요하다.)

createMenus()함수는 처음에 호출된 createLanguageMenu()를 참고하여 Language차림표에 언어목록을 채운다.

```
void MainWindow::createLanguageMenu()
{
    QDir dir(qmPath);
    QStringList fileNames = dir.entryList("callcenter_*.qm");
    for (int i = 0;i < (int)fileNames.size();++i) {
        QTranslator translator;
        translator.load(fileNames[i], qmPath);
        QTranslatorMessage message = translator.findMessage("MainWindow", "English");
        QString language = message.translation();
        int id = languageMenu->insertItem( tr("&%1 %2")
                        .arg(i + 1).arg(language), this, SLOT(switchToLanguage(int)));
        languageMenu->setItemParameter(id, i);
        if (language == "English")
                languageMenu->setItemChecked(id, true);
        QString locale = fileNames[i];
        locale = locale.mid(locale.find('_') + 1);
        locale.truncate(locale.find('.'));
        locales.push_back(locale);
    }
}
```

응용프로그람이 유지하는 언어들을 코드작성하지 않고 응용프로그람의 translations등록부에 배치된 매개 .qm파일용으로 차림표항목을 하나씩 창조한다. 단순히 영어용의 .qm파일이 있다고 가정한다. 사용자가 영어를 선택할 때 QTranslator객체들에 대하여 clear()를 선택하는 방법이 있다.

한가지 특별한 난관은 매개 .qm파일이 제공하는 언어용으로 좋은 이름을 제시하는것이

다. .qm파일의 이름에 기초하여 English를 en으로 혹은 Deutsch를 de로 표시하면 미숙해보이며 일부 사용자들은 혼돈한다. createLanguageMenu()에서 사용한 방법은 MainWindow문맥에서 문자렬 "English"의 번역을 검사하는것이다. 그 문자렬은 도이칠란드어번역에서 "Deutsch"로, 프랑스어번역에서 "Français"로, 일본어번역에서 "日本語"로 번역되여야 한다.

QPopupMenu::insertItem()를 리용하여 차림표항목들을 창조한다. 이것들은 모두 기본창문의 switchToLanguage(int)처리부에 련결되는데 다음에 고찰한다. switchToLanguage(int)처리부의 파라메터는 setItemParameter()에 의해 설정한 값이다. 이것은 3장에서 표계산프로그람의 최근에 연 파일목록을 실현할 때 수행한것과 아주 비슷하다.

끝으로 switchToLanguage()을 실현하는데 사용하는 locales라고 부르는 QStringList에 지역을 추가한다.

```
void MainWindow::switchToLanguage(int param)
{
    appTranslator->load("callcenter_" + locales[param], qmPath);
    qtTranslator->load("qt_" + locales[param], qmPath);
    for (int i = 0; i < (int)languageMenu->count(); ++i)
        languageMenu->setItemChecked(languageMenu->idAt(i), i == param);
    retranslateStrings();
}
```

switchToLanguage()처리부는 사용자가 Language차림표로부터 언어를 선택할 때 호출된다. 응용프로그람과 Qt용의 번역파일을 적재하는것으로 시작한다. 그다음 Language차림표항목앞의 검사표식들을 갱신하여 사용중에 있는 언어를 표식하고 retranslateStrings()를 호출하여 기본창문의 문자렬들을 모두 재번역한다.

Microsoft Windows에서 Language차림표를 제공하는 다른 수법은 환경의 지역에서 변화를 탐지할 때 Qt에 의해 발생된 사건형인 LocaleChange사건에 응답하는것이다. 사건형은 Qt에 유지된 모든 가동환경들에서 존재하지만 Windows에서는 사용자가 체계의 지역설정(Control Panel의 Regional and Language Options)을 변경할 때 실제로 생성된다. LocaleChange사건들을 조종하기 위하여 QObject::event()를 다음과 같이 재정의한다.

```
bool MainWindow::event(QEvent *event)
{
    if (event->type() == QEvent::LocaleChange) {
        appTranslator->load(QString("callcenter_") + QTextCodec::locale(), qmPath);
        qtTranslator->load(QString("qt_") + QTextCodec::locale(), qmPath);
        retranslateStrings();
    }
    return QMainWindow::event(event);
```

```
}
```

응용프로그람이 실행중에 있을 때 사용자가 지역을 절환하면 새 지역용의 정확한 번역파일들을 적재하고 retranslateStrings()를 호출하여 사용자대면부를 갱신한다.

모든 경우에 기초클라스중 하나가 LocaleChange사건들과 관련될수도 있으므로 기초클라스의 event()함수에 대하여 사건을 넘긴다.

이것으로 MainWindow코드의 고찰을 끝낸다. 그러면 응용프로그람의 창문부품클라스들중 하나인 JournalView클라스의 코드를 고찰하여 동적번역을 유지하려면 어떤 변경이 필요한가를 고찰해보자.

```
JournalView::JournalView(QWidget *parent, const char *name) : QListView(parent, name)
{
    ...
    retranslateStrings();
}
```

JournalView클라스는 QListView의 파생클라스이다. 구성자의 끝에서 비공개함수 retranslateStrings()를 호출하여 창문부품의 문자렬들을 설정한다. 이것은 MainWindow에서 수행한것과 비슷하다.

```
bool JournalView::event(QEvent *event)
{
    if (event->type() == QEvent::LanguageChange)
        retranslateStrings();
    return QListView::event(event);
}
```

event()함수를 재정의하여 LanguageChange사건들에 대하여 retranslateStrings()를 호출한다.

Qt는 QApplication에 현재 설치된 QTranslator의 내용이 변할 때 LanguageChange사건을 생성한다. Call Center응용프로그람에서 이것은 MainWindow::switchToLanguage()나 MainWindow::event()로부터 appTranslator 혹은 qtTranslator에 대하여 load()를 호출할 때 발생한다.

LanguageChange사건들은 LocaleChange사건들과 같지 않다. LocaleChange사건은 응용프로그람에 "Maybe you should load a new translation."라고 알린다. 대조적으로 LanguageChange사건은 응용프로그람의 창문부품들에 "Maybe you should retranslate all your strings."라고 알린다.

MainWindow를 실현했을 때 LanguageChange에 응답할 필요가 없었다. 그대신에 QTranslator에 대하여 load()를 호출했을 때마다 단순히 retranslateStrings()를 호출하였다.

```
void JournalView::retranslateStrings()
{
    for (int i = columns() -1; i >= 0; --i)
        removeColumn(i);
```

```
    addColumn(tr("Time"));
    addColumn(tr("Priority"));
    addColumn(tr("Phone Number"));
    addColumn(tr("Subject"));
}
```

retranslateStrings()함수는 새로 번역된 본문을 가지고 QListView의 렬제목들을 다시 창조한다. 모든 렬제목들을 삭제하고 새로운 렬제목들을 추가하고 이것을 수행한다. 이 조작은 오직 QListView제목에만 영향을 주며 QListView에 보관한 자료에는 영향을 주지 않는다.

이것으로 손으로 쓴 창문부품의 번역관련코드의 설명을 끝낸다. Qt Designer로 개발한 창문부품과 대화칸들에서 uic도구는 LanguageChange사건들에 응답하여 자동적으로 호출되는 retranslateStrings()함수와 비슷한 함수를 자동적으로 생성한다. 우리가 해야 할 일은 사용자가 언어를 절환할 때 번역파일을 적재하는것이다.

제4절. 응용프로그람의 번역

tr()호출을 포함하는 Qt응용프로그람의 번역은 3단계의 과정으로 되여있다.

① lupdate를 실행하여 사용자에게 표시하는 문자렬들을 응용프로그람의 원천코드에서 얻는다.

② Qt Linguist에 의하여 응용프로그람을 번역한다.

③ lrelease을 실행하여 응용프로그람이 QTranslator를 사용하여 적재하는 2진.qm파일을 생성한다.

걸음 ①과 ③은 응용프로그람개발자들이 수행한다. 걸음 ②는 번역기가 처리한다. 이 주기는 응용프로그람의 개발과 수명기간 필요할 때마다 반복될수 있다.

실례로 3장의 표계산프로그람을 번역하는 방법을 보기로 한다. 응용프로그람은 이미 사용자에게 표시하는 문자렬주위에 tr()호출을 포함한다.

우선 응용프로그람의 .pro파일을 수정하여 유지하려는 언어들을 지정해야 한다. 례를 들면 영어와 함께 도이췰란드어와 프랑스어를 유지하려고 한다면 다음의 TRANSLATIONS항목을 spreadsheet.pro에 추가해야 한다.

```
TRANSLATIONS = spreadsheet_de.ts \ spreadsheet_fr.ts
```

여기서는 2개의 번역파일 즉 도이췰란드어파일과 프랑스어파일을 지정한다. 이 파일들은 처음으로 lupdate를 실행할 때 창조되고 후에 lupdate를 실행할 때마다 갱신된다.

보통 이 파일들은 .ts확장자를 가진다. 이것들은 간단한 XML형식으로 되여있고 QTranslator에 의해 해석된 2진.qm파일들처럼 조밀하지 않다. 사람이 읽을수 있는 .ts파일을 콤퓨터에 효과적인 .qm파일들로 변환하는것은 lrelease의 일감이다. 자세히 말하면 .ts는 번역원천파일을 의미하고 .qm은 Qt통보문파일을 의미한다.

우리가 표계산프로그람의 원천코드를 포함한 등록부안에 있다고 가정하면 spreadsheet.pro에 대하여 지령행에서 lupdate를 다음과 같이 실행할수 있다.

lupdate -verbose spreadsheet.pro

-verbose인수는 선택인수이다. 이것은 lupdate에 보통보다 반결합을 더 제공한다는것을 알린다. 여기에 기대한 출력이 있다.

Updating 'spreadsheet_de.ts'...

0 known, 101 new and 0 obsoleted messages

Updating 'spreadsheet_fr.ts'...

0 known, 101 new and 0 obsoleted messages

응용프로그람원천코드의 tr()호출안에서 나타나는 모든 문자렬은 빈 번역을 가지는 .ts파일들에 보관된다. 응용프로그람의 .ui파일들에 나타나는 문자렬들도 포함된다.

lupdate도구는 기정으로 tr()인수들이 Latin-1문자렬이라고 가정한다. 그렇지 않으면 CODEC 항목을 .pro파일에 추가해야 한다. 례를 들면

CODEC = EUC-JP

이것은 응용프로그람의 main()함수로부터 QTextCodec::setCodecForTr()호출과 함께 수행되여야 한다.

그다음 Qt응용프로그람들을 번역하는 GUI도구인 Qt Linguist를 리용하여 spreadsheet_de.ts와 spreadsheet_fr.ts파일에 번역을 추가해야 한다.

Qt Linguist를 기동하려면 Windows에서는 Start차림표에서 Qt 3.2.x|Qt Linguist를 찰칵하고 Unix에서는 지령행에서 linguist라고 입력하며 Mac OS X Finder에서는 linguist를 두번 찰칵한다. .ts파일에 대한 번역추가를 시작하려면 File|Open를 찰칵하고 파일을 선택한다.

Qt Linguist기본창문의 왼쪽에 번역중에 있는 응용프로그람의 문맥목록들이 표시된다. 표계산프로그람에서 문맥들로서는 FindDialog, GoToCellDialog, MainWindow, SortDialog, Spreadsheet가 있다. 오른쪽웃구역은 현재 문맥용의 원천본문목록이다. 매개 원천본문은 번역과 *Done*기발과 함께 표시된다. 오른쪽 중간구역은 현재 원천항목의 번역을 입력할수 있는 곳이다. 오른쪽 아래구역은 Qt Linguist가 자동적으로 제공하는 제안목록이다.

번역된 .ts파일을 일단 얻으면 그것을 2진.qm파일로 변환하여 QTranslator가 리해할수 있게 해야 한다. 그러자면 Qt Linguist안에서 File|Release를 찰칵한다. 일반적으로 일부 문자렬만 번역하는것으로 시작하며 .qm파일을 가지고 응용프로그람을 실행하여 제대로 작업하는가 확인한다.

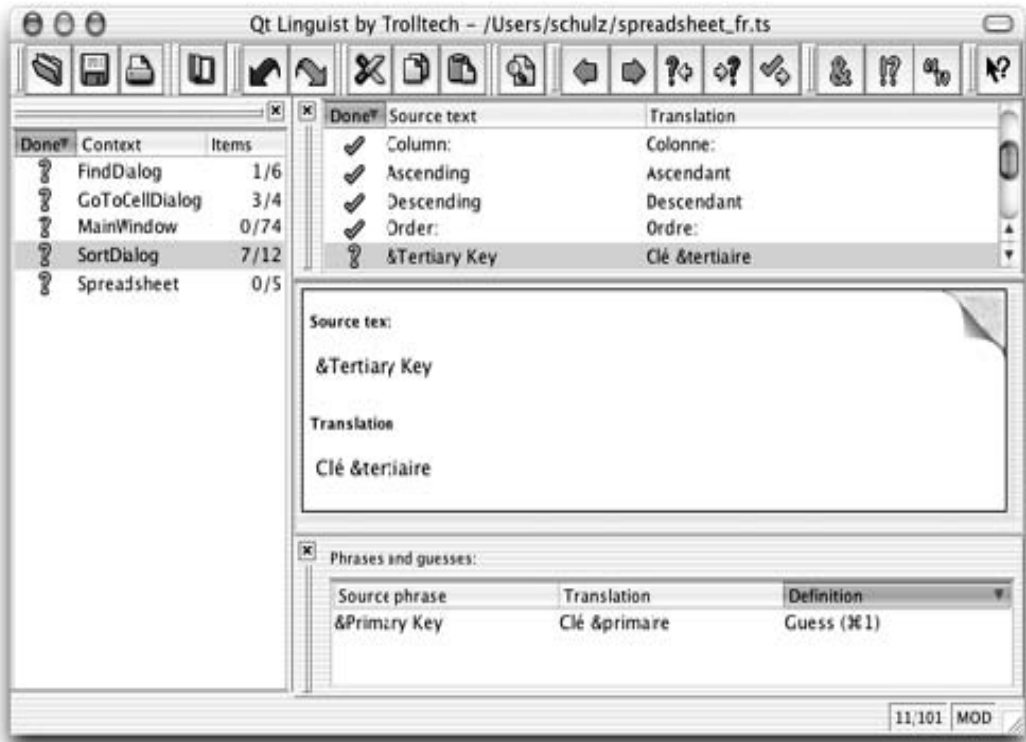



그림 15-2. Qt Linguist의 작용

모든 .ts파일들에 대하여 .qm파일들을 다시 생성하려고 한다면 lrelease지령행도구를 다음과 같이 사용한다.

lrelease -verbose spreadsheet.pro

19개 문자렬을 프랑스어로 번역하고 그중 17개에 Done기발을 찰칵하였다고 가정하면 lrelease는 다음의 출력을 생성한다.

```
Updating 'spreadsheet_de.qm'...
0 finished, 0 unfinished and 101 untranslated messages
Updating 'spreadsheet_fr.qm'...
17 finished, 2 unfinished and 82 untranslated messages
```

번역하지 않은 문자렬들은 응용프로그람을 실행할 때 원래언어로 표시된다. Done기발은 lrelease가 사용하지 않으며 번역기들이 어느 번역을 끝내고 어느 번역을 다시 방문해야 하는가를 식별하는데 쓰인다.

응용프로그람의 원천코드를 수정할 때 번역파일들이 갱신될수 있다. 해결책은 lupdate를 다시 실행하고 새 문자렬들의 번역을 제공하고 .qm파일들을 다시 생성하는것이다. 일부 개발팀들은 lupdate를 자주 실행하기 좋아하며 다른 팀들은 최종제품을 출하하기 직전까지 기다리기를 좋아한다.

lupdate와 Qt Linguist도구는 아주 고급한 도구이다. 더는 사용하지 않는 번역은 후에 출하할 때 요구되는 경우에만 .ts파일들에 보관한다. .ts파일들을 갱신할 때 lupdate는 지능결합알고

리듬을 사용하므로 번역기들이 각이한 문맥들에서 같거나 류사한 본문을 번역할 때 상당한 시간을 절약할수 있게 한다.

제16장. 직결방조의 제공

대부분의 응용프로그람들은 사용자에게 직결방조를 제공한다. 어떤 방조는 도구암시와 상태암시, What's This?방조와 같이 간단하다. Qt는 이것들을 모두 제공한다. 다른 방조는 훨씬 범위가 넓고 많은 본문페지를 요구한다. 이러한 종류의 방조에서는 QTextBrowser를 단순한 직결방조열람기로 사용하거나 자기의 응용프로그람으로부터 Qt Assistant 혹은 다른 HTML열람기를 펼칠수 있다.

제1절. 도구암시와 상태암시, What's This?방조

도구암시는 일정한 시간동안 창문부품우에서 마우스가 머무를 때 나타나는 자그마한 본문부분이다. 도구암시는 황색배경에 흑색본문으로 표시된다. 그 주되는 용도는 도구띠단추들에 본문설명을 제공하는것이다.

QToolTip::add()를 사용하여 임의의 창문부품들에 도구암시를 추가할수 있다. 례를 들면

```
QToolTip::add(findButton, tr("Find next"));
```

QAction에 대응하는 도구띠단추의 도구암시를 설정하려면 간단히 그 작용에 대하여 setToolTip()를 호출해야 한다. 례를 들면

```
newAct = new QAction(tr("&New"), tr("Ctrl+N"), this);
newAct->setToolTip(tr("New file"));
```

도구암시를 명시적으로 설정하지 않으면 QAction은 작용본문과 지름건(례를 들면 "New (Ctrl+N)")으로부터 자동적으로 생성한다.

상태암시도 역시 짧은 설명본문으로서 보통 도구암시보다 좀 길다. 마우스가 도구띠단추나 차림표선택우에 머무를 때 상태암시가 상태띠에 나타난다. setStatusTip()를 호출하여 작용에 상태암시를 추가한다.

```
newAct->setStatusTip(tr("Create a new file"));
```

상태암시가 없을 때 QAction은 그대신에 도구암시본문을 사용한다.

QAction을 사용하지 않으면 QToolTipGroup객체와 상태암시를 QToolTip::add()의 셋째와 넷째 인수로서 넘겨야 한다.

```
QToolTip::add(findButton, tr("Find next"), toolTipGroup,
              tr("Find the next occurrence of the search text"));
```

응용프로그람은 QToolTipGroup의 showTip()와 removeTip()신호들을 상태띠의 message()와 clear()처리부들에 련결함으로써 상태띠에 긴 본문을 표시할수 있다. QToolTipGroup객체는 긴 방조본문을 표시할수 있는 도구암시들과 창문부품사이의 교제를 관리할수 있는 응답성이 있어야 한다.

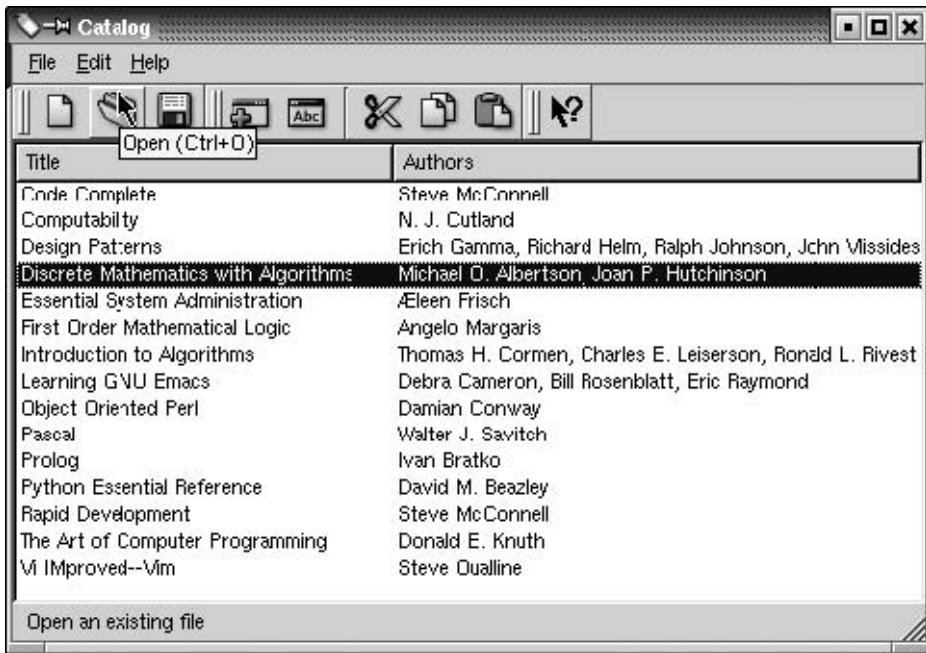



그림 16-1. 도구암시와 상태암시를 표시하는 응용프로그람

*Qt Designer*에서 도구암시와 상태암시는 창문부품이나 작용의 도구암시와 statusTip속성들을 통하여 호출할수 있다.

일부 상황에서는 창문부품에 대하여 도구암시나 상태암시로 줄수 있는것보다 많은 정보를 제공해야 한다. 례를 들면 복잡한 대화칸에서 개별적인 방조창문을 사용자가 펼치지 않고 각 마당에 대한 설명본문을 제공해야 한다. What's This?방식은 이를 위한 리상적인 해결책이다. 창문이 What's This?방식에 있을 때 유표는 로 변경되고 사용자는 임의의 사용자대면부부분품을 찰칵하여 방조본문을 얻을수 있다. What's This?방식에 들어가기 위하여 사용자는 대화칸의 제목띠(Windows와 KDE)의 *?*단추를 찰칵하거나 *Shift*+F1를 누를수 있다.

방조본문은 QWhatsThis::add()를 호출하여 설정할수 있다. 여기에 실례가 있다.

```
QWhatsThis::add(sourceLineEdit,
        tr("<img src=\"icon.png\">"
           " The meaning of the Source field depends on the Type field:"
           "<ul>"
              "<li><b>Books</b> have a Publisher</li>"
              "<li><b>Articles</b> have a Journal name with volume and issue number</li>"
              "<li><b>Thesis</b> have an Institution name and a department name</li>"
           "</ul>"));
```

다른 많은 Qt창문부품들처럼 HTML형식꼬리표들을 사용하여 도구암시의 본문을 형식화할수 있다. 실례에서는 화상(응용프로그람의 .pro파일에서 IMAGE항목에 렬거된다.), 게시목록, 강조체의 본문을 포함한다. Qt가 유지하는 꼬리표들은 QStyleSheet문서에 지정된다.

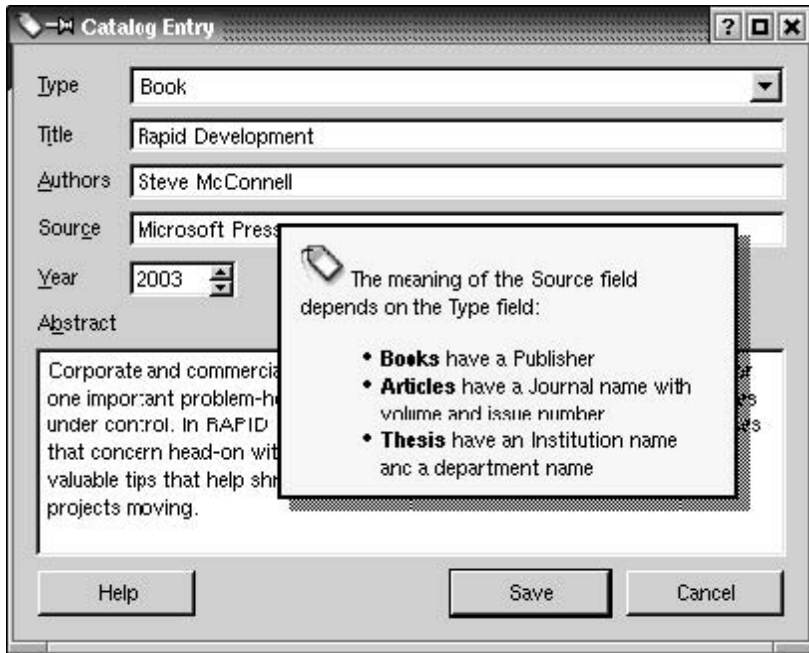



그림 16-2. What's This?방조본문을 표시하는 대화칸

또한 작용에 대하여 What's This?본문을 설정할수 있다.

```
openAct->setWhatsThis(tr("<img src=open.png> "
        "Click this option to open an existing file."));
```

본문은 What's This?방식에 있을 때 사용자가 차림표항목이나 도구띠단추를 찰칵하거나 지름건을 누를 때 표시된다. Qt Designer에서 창문부품이나 작용의 What's This?본문은 whatsThis속성을 통하여 사용할수 있다.

응용프로그람기본창문의 사용자대면부부분품들은 What's This?본문을 제공할 때 What's This?도구띠단추는 물론 Help차림표에 What's This?선택을 제공하는것이 관례로 되여있다. 이것은 What's This?작용을 창조하고 그 activated()신호를 실행시에 What's This?방식에 들어가는 QMainWindow의 whatsThis()처리부에 련결하여 수행한다.

제2절. 단순한 방조엔진 QTextBrowser의 사용

크고 복잡한 응용프로그람들은 도구암시, 상태암시, What's This?방조가 제공하는것보다 더 많은 직결방조를 요구한다. 간단한 해결책은 방조열람기를 제공하는것이다. 일반적으로 방조열람기를 제공하는 응용프로그람들은 기본창문의 Help차림표에 Help항목을 모든 대화칸에 Help단추를 가지고있다.

이 절에서는 그림 16-3에 보여주는 간단한 방조열람기를 표시하고 응용프로그람에서 그 사용방법을 설명한다. 창문은 QTextBrowser를 사용하여 HTML기초문법으로 표식되는 방조페지들을 실현한다. QTextBrowser는 수많은 단순한 HTML꼬리표들을 처리할수 있으므로 이러한 목적에 리상적이다.

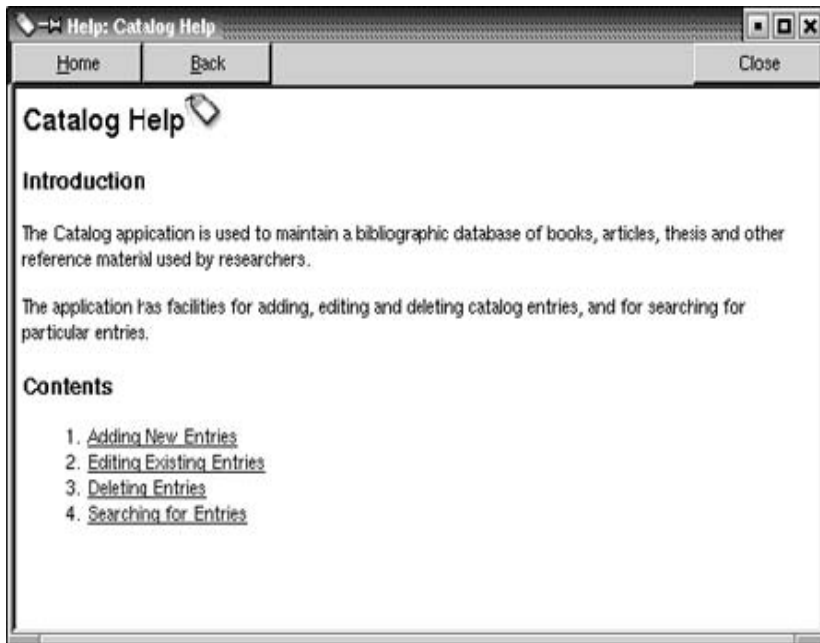



그림 16-3. HelpBrowser창문부품

머리부파일부터 시작하자.

```
#include <qwidget.h>
class QPushButton;class QTextBrowser;
class HelpBrowser : public QWidget
{
    Q_OBJECT
public:
    HelpBrowser(const QString &path, const QString &page, QWidget *parent = 0,
                const char *name = 0);
    static void showPage(const QString &page);
private slots:
    void updateCaption();
private:
    QTextBrowser *textBrowser;
    QPushButton *homeButton;
    QPushButton *backButton;
    QPushButton *closeButton;
};
```

HelpBrowser는 응용프로그람의 어디서나 호출할수 있는 정적함수를 제공한다. 이 함수는 HelpBrowser창문을 창조하고 주어진 페지를 표시한다.

여기에 실현의 앞부분이 있다.

```
#include <qapplication.h>
#include <qlayout.h>
#include <qpushbutton.h>
#include <qtextbrowser.h>
#include "helpbrowser.h"
HelpBrowser::HelpBrowser(const QString &path, const QString &page, QWidget *parent,
    const char *name) : QWidget(parent, name, WGroupLeader | WDestructiveClose)
{
    textBrowser = new QTextBrowser(this);
    homeButton = new QPushButton(tr("&Home"), this);
    backButton = new QPushButton(tr("&Back"), this);
    closeButton = new QPushButton(tr("&Close"), this);
    closeButton->setAccel(tr("Esc"));
    QVBoxLayout *mainLayout = new QVBoxLayout(this);
    QHBoxLayout *buttonLayout = new QHBoxLayout(mainLayout);
    buttonLayout->addWidget(homeButton);
    buttonLayout->addWidget(backButton);
    buttonLayout->addStretch(1);
    buttonLayout->addWidget(closeButton);
    mainLayout->addWidget(textBrowser);
    connect(homeButton, SIGNAL(clicked()), textBrowser, SLOT(home()));
    connect(backButton, SIGNAL(clicked()), textBrowser, SLOT(backward()));
    connect(closeButton, SIGNAL(clicked()), this, SLOT(close()));
    connect(textBrowser, SIGNAL(sourceChanged(const QString &)), this,
                        SLOT(updateCaption()));
    textBrowser->mimeSourceFactory()->addFilePath(path);
    textBrowser->setSource(page);
}
```

배치에는 단순히 QTextBrowser의 제일 우의 한행에 놓인 단추들이 포함된다. path파라메터는 응용프로그람의 문서를 포함하는 파일체계안의 경로이다. page파라메터는 문서파일의 이름인데 선택적인 HTML닻(anchor)을 가지고있다.

기본창문과 함께 이행금지대화칸으로부터 HelpBrowser창문들을 펼치려고 하므로 WGroupLeader기발을 사용한다. 보통 이행금지대화칸은 사용자가 응용프로그람의 다른 창문과 교제하는것을 방지한다. 그러나 방조요구후에 사용자는 명백히 이행금지대화칸와 방조열람기

모두와 교제할수 있게 되여야 한다. WGroupLeader기발의 사용은 이 교제를 가능하게 한다.

```
void HelpBrowser::updateCaption()
{
    setCaption(tr("Help: %1") .arg(textBrowser->documentTitle()));
}
```

원천페지가 달라질 때마다 updateCaption()처리부가 실행된다. documentTitle()함수는 페지의 <title>꼬리표에 지정한 본문을 돌려준다.

```
void HelpBrowser::showPage(const QString &page)
{
    QString path = qApp->applicationDirPath() + "/doc";
    HelpBrowser *browser = new HelpBrowser(path, page);
    browser->resize(500, 400);
    browser->show();
}
```

showPage()정적함수에서는 HelpBrowser창문을 창조하고 그것을 표시한다. 구성자에서 WDestructiveClose기발을 설정하였으므로 창문은 사용자가 닫을 때 자동적으로 해체된다.

이 실례에서는 응용프로그람의 실행파일을 포함하는 등록부의 doc보조등록부에 문서가 배치된다고 가정한다. showPage()함수에 넘긴 모든 페지는 이 doc보조등록부로부터 얻는다.

이제는 응용프로그람으로부터 방조열람기를 펼칠 준비가 되였다. 응용프로그람의 기본창문에서는 *Help*작용을 창조하고 그것을 다음과 같은 help()처리부에 련결한다.

```
void MainWindow::help()
{
    HelpBrowser::showPage("index.html");
}
```

이것은 기본방조파일이 index.html로부터 호출된다고 가정한다. 대화칸들에서는 *Help*단추를 다음과 같은 help()처리부에 련결한다.

```
void EntryDialog::help()
{
    HelpBrowser::showPage("dialogs.html#entrydialog");
}
```

여기서는 다른 방조파일인 dialogs.html에 있으며 QTextBrowser를 entrydialog닻으로 흘림한다.

방조를 펼칠수 있는 또 다른 위치는 What's This?본문이다. HTML <a href="...">꼬리표들을 사용하여 What's This?본문을 문서에 련결할수 있다.

The meaning of the Source field depends on the Type field:

- **Books** have a Publisher
- **Articles** have a Journal name with volume and issue number
- **Thesis** have an Institution name and a department name

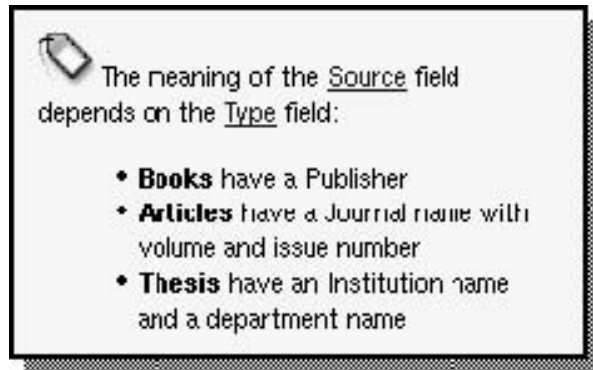



그림 16-4. 련결을 가진 What's This?본문

What's This?본문으로부터 초본문련결이 동작하게 하기 위하여 방조열람기를 알고있는 QWhatsThis를 사용해야 한다. 이것은 QWhatsThis의 파생클라스를 만들고 그의 clicked()함수를 재정의하고 HelpBrowser::showPage()를 호출하여 달성한다. 여기에 클라스정의가 있다.

```
class MyWhatsThis : public QWhatsThis
{
public:
    MyWhatsThis(QWidget *widget, const QString &text);
    QString text(const QPoint &point);
    bool clicked(const QString &page);
private:
    QString myText;
};
```

text()와 clicked()함수는 QWhatsThis로부터 재정의된다.

```
MyWhatsThis::MyWhatsThis(QWidget *widget, const QString &text)
     : QWhatsThis(widget) { myText = text;}
```

구성자는 창문부품과 그 창문부품의 What's This?본문을 받아들인다. 창문부품을 기초클라스에 넘기고 본문을 비공개변수에 보관한다.

```
QString MyWhatsThis::text(const QPoint &) { return myText;}
```

text()함수는 마우스유표위치가 주어진 창문부품용의 What's This?본문을 돌려준다. 일부 창문부품들에서는 사용자가 찰칵한 위치에 따라 각이한 본문을 돌려주어야 하지만 여기서는 늘 같은 본문을 돌려준다.

```
bool MyWhatsThis::clicked(const QString &page)
{
    if (page.isEmpty()) {
       return true;
    } else {
```

```
            HelpBrowser::showPage(page);
            return false;
        }
    }
```

clicked()함수는 사용자가 What's This?창문우에서 찰칵할 때 QWhatsThis에 의하여 호출된다. 사용자가 HTML련결을 찰칵했을 때 QWhatsThis는 목표페지를 clicked()함수에 넘긴다. (아무것도 찰칵하지 않았으면 빈 문자렬을 넘긴다.) 방조열람기를 펼치고 주어진 페지를 표시한다.

clicked()의 돌림값은 QWhatsThis가 What's This?본문을 숨겨야 하는가(true로 지적한다.) 계속 표시하여야 하는가를 결정하는데 사용된다. 사용자가 련결을 찰칵할 때 What's This?가 방조창문에 계속 표시되기를 바라므로 false를 돌려준다. 사용자가 What's This?창문안의 어떤 곳을 찰칵하면 true를 돌려주어 What's This?창문을 숨긴다.

여기에 MyWhatsThis클라스를 사용하는 방법이 있다.

```
    new MyWhatsThis(sourceLineEdit,
    tr("<img src=\"icon.png\">"
        " The meaning of the "
        "<a href=\"fields.html#source\">Source</a> field depends on "
        "the <a href=\"fields.html#type\">Type</a> field:"
        "<ul>"
           "<li><b>Books</b> have a Publisher</li>"
           "<li><b>Articles</b> have a Journal name with volume and issue number</li>"
           "<li><b>Thesis</b> have an Institution name and a department name</li>"
        "</ul>"));
```

QWhatsThis::add()을 호출하지 않고 창문부품과 그 련관본문을 가지는 MyWhatsThis객체를 창조한다. 그러나 이번에 사용자가 련결을 찰칵하면 방조열람기가 펼쳐진다.

new를 리용하여 객체를 할당하고 변수에 값을 대입하지 않는다. 이것은 Qt가 모든 QWhatsThis객체들의 상태를 보관하고있다가 그것들이 더는 요구되지 않을 때 삭제하면 여기서 문제로 되지 않는다.

제3절. 강력한 직결방조 Qt Assistant의 사용

Qt Assistant는 Trolltech가 제공하는 재배포가능한 직결방조프로그람이다. 그 주요한 우점은 색인화와 완전본문탐색을 유지하고 응용프로그람들에서 여러개의 문서모임들을 처리할수 있는것이다.

Qt Assistant를 사용하려면 응용프로그람에 필요한 코드를 통합하여 Qt Assistant가 문서를 알게 하여야 한다.

Qt응용프로그람과 Qt Assistant사이의 교제는 개별적인 서고에 배치된 QAssistantClient에 의해 처리된다. 서고를 응용프로그람과 련결하려면 다음 행을 응용프로그람의 .pro파일에 추가

해야 한다.

```
LIBS += -lqassistantclient
```

그러면 *Qt Assistant*를 사용하는 새로운 HelpBrowser클라스의 코드를 고찰하자.

```
#ifndef HELPBROWSER_H
#define HELPBROWSER_H
class QAssistantClient;
class HelpBrowser
{
public:
    static void showPage(const QString &page);
private:
    static QAssistantClient *assistant;
};
#endif
```

여기에 새로운 helpbrowser.cpp가 있다.

```
#include <qassistantclient.h>
#include "helpbrowser.h"
QAssistantClient *HelpBrowser::assistant = 0;
void HelpBrowser::showPage(const QString &page)
{
    if (!assistant)
          assistant = new QAssistantClient("");
    assistant->showPage(page);
}
```

QAssistantClient구성자는 경로문자렬을 첫 인수로 받아들인다. 이것은 Qt Assistant실행파일을 탐색하는데 사용된다. 빈 경로를 넘기면 QAssistantClient가 PATH환경변수안에서 실행파일을 찾는다.

QAssistantClient는 처음에 QTextBrowser파생클라스의 showPage()함수와 같이 선택적인 HTML닻을 가지는 페지이름을 받아들이는 자체의 showPage()함수를 가지고있다.

다음 단계는 Qt Assistant에게 문서가 있는 위치를 알리는것이다. 이것은 Qt Assistant프로파일을 창조하고 문서에 대한 정보를 제공하는 .dcf파일을 창조함으로써 수행한다. 이것은 모두 Qt참고자료(도구편)의 3장에서 설명한다.

QTextBrowser나 Qt Assistant를 사용하는 다른 방법은 가동환경에 고유한 수법으로 직결방조를 제공하는것이다. Windows응용프로그람들에서는 Windows HTML방조파일들을 창조하고 Microsoft Internet Explorer에 의하여 그것들을 호출할수 있다. Qt의 QProcess클라스 혹은

ActiveQt틀거리를 사용할수 있다. Unix와 Mac OS X응용프로그람들에서 적합한 수법은 HTML 파일들을 제공하여 웨브열람기를 펼치는것이다.

제17장. 다중스레드작성

보통의 GUI응용프로그람들은 하나의 실행스레드를 가지며 한번에 하나의 조작을 수행한다. 사용자가 단일스레드 응용프로그람에서 사용자대면부로부터 시간을 소비하는 조작을 호출하면 그 조작이 처리되는동안 대면부는 일반적으로 동결된다. 7장(사건처리)은 이 문제에 대한 대책을 제공하며 다중스레드작성은 또 하나의 대책으로 된다.

다중스레드 Qt응용프로그람에서 GUI는 자기의 스레드에서 실행되며 처리는 하나이상의 다른 스레드들에서 진행된다. 이것은 긴장한 처리를 하는 경우에도 응용프로그람들이 제각기 GUI를 가지게 한다. 다중스레드작성의 다른 하나의 리득은 다중처리소자콤퓨터들에서 서로 다른 스레드들을 동시에 각이한 처리소자들에서 실행하여 좋은 성능을 낼수 있다는것이다.

이 장에서는 우선 QThread의 파생클라스를 만드는 방법과 스레드들을 동기시키는데 QMutex, QSemaphore, QWaitCondition를 사용하는 방법을 보여준다. 그다음 사건순환고리를 실행하는동안 비GUI스레드들에서 GUI스레드와 교제하는 방법을 보여주고 어느 Qt클라스들을 비GUI스레드들에서 사용할수 있고 어느것을 사용할수 없는가 고찰한다.

다중스레드작성은 큰 주제로서 이것을 포괄적으로 설명하는 도서들은 많다. 여기서는 다중스레드프로그람작성의 기초를 리해하고있는것을 전제로 하고 스레드작성 그 자체보다도 다중스레드 Qt응용프로그람개발방법을 설명하는데 초점을 둔다.

제1절. 스레드와의 작업

Qt응용프로그람에서 다중스레드를 제공하는것은 간단하다. 즉 QThread의 파생클라스를 만들고 그의 run()함수를 재정의한다. 그 과정을 보여주기 위하여 우선 콘솔에 같은 본문을 반복 출력하는 아주 간단한 QThread의 파생클라스코드를 고찰한다.

```
class Thread : public QThread
{
public:
    Thread();
    void setMessage(const QString &message);
    void run();
    void stop();
private:
    QString messageStr;
    volatile bool stopped;
};
```

Thread클라스는 QThread를 계승하며 run()함수를 재정의한다. 이 클라스는 2개의 추가함수 setMessage()와 stop()을 제공한다.

stopped변수는 각이한 스레드들로부터 호출되고 필요할 때마다 새로 읽어들이는것을 확인하려고 하므로 volatile로 선언한다. volatile예약어를 생략하면 콤파일러는 변수호출을 최적화할수 있으나 부정확한 결과를 초래할수 있다.

```
Thread::Thread()
{
    stopped = false;
}
```

구성자에서 stopped를 false로 설정한다.

```
void Thread::run()
{
    while (!stopped)
        cerr << messageStr.ascii();
    stopped = false;
    cerr << endl;
}
```

run()함수가 호출되면 스레드실행이 시작된다. stopped변수가 false인 경우에 함수는 콘솔에 주어진 통보문을 출력한다. 스레드는 조종이 run()함수를 벗어날 때 완료한다.

```
void Thread::stop()
{
    stopped = true;
}
```

stop()함수는 stopped변수를 true로 설정하여 run()이 콘솔에 대한 본문출력을 정지하게 한다. 이 함수는 임의의 스레드로부터 임의의 시간에 호출할수 있다. 실례에서는 bool에 대한 대입이 원자연산이라고 가정한다. 이것은 타당한 가정으로서 bool이 true나 false라는것을 고려한다. 이 절에서 후에 QMutex를 사용하여 변수에 대한 대입이 원자연산이라는것을 담보하는 방법을 알게 된다.

QThread는 실행중에 있는 스레드의 실행을 완료하는 terminate()함수를 제공한다. terminate()의 사용은 권고하지 않는다. 그것은 임의의 점에서 스레드를 정지시킬수 있고 후에 스레드에 그자체를 삭제할 기회를 주지 않기때문이다. 여기서 수행하는것처럼 stopped변수와 stop()함수를 사용하는것이 늘 안전하다.

그림 17-1. Threads응용프로그람

이제는 처음의 스레드와 함께 2개의 스레드 A와 B를 사용하는 자그마한 Qt응용프로그람에서 Thread클라스를 사용하는 방법을 고찰한다.

```
class ThreadForm : public QDialog
{
    Q_OBJECT
public:
    ThreadForm(QWidget *parent = 0, const char *name = 0);
protected:
    void closeEvent(QCloseEvent *event);
private slots:
    void startOrStopThreadA();void startOrStopThreadB();
private:
    Thread threadA;
    Thread threadB;
    QPushButton *threadAButton;
    QPushButton *threadBButton;
    QPushButton *quitButton;
};
```

ThreadForm클라스는 Thread형의 2개 변수와 단추들을 선언하며 기본사용자대면부를 제공한다.

```
ThreadForm::ThreadForm(QWidget *parent, const char *name) : QDialog(parent, name)
{
    setCaption(tr("Threads"));
    threadA.setMessage("A");
    threadB.setMessage("B");
    threadAButton = new QPushButton(tr("Start A"), this);
    threadBButton = new QPushButton(tr("Start B"), this);
    quitButton = new QPushButton(tr("Quit"), this);
    quitButton->setDefault(true);
    connect(threadAButton, SIGNAL(clicked()), this, SLOT(startOrStopThreadA()));
    connect(threadBButton, SIGNAL(clicked()), this, SLOT(startOrStopThreadB()));
    connect(quitButton, SIGNAL(clicked()), this, SLOT(close()));
    ...
}
```

구성자에서는 setMessage()을 호출하여 첫 스레드는 "A"를, 둘째 스레드는 "B"를 반복 출

력하게 한다.

```
void ThreadForm::startOrStopThreadA()
{
    if (threadA.running()) {
        threadA.stop();
        threadAButton->setText(tr("Start A"));
    } else {
        threadA.start();
        threadAButton->setText(tr("Stop A"));
    }
}
```

사용자가 스레드 A용의 단추를 찰칵할 때 startOrStopThreadA()는 스레드가 실행중에 있으면 정지하고 그렇지 않으면 스레드를 기동한다. 또한 단추의 본문을 갱신한다.

```
void ThreadForm::startOrStopThreadB()
{
    if (threadB.running()) {
        threadB.stop();
        threadBButton->setText(tr("Start B"));
    } else {
        threadB.start();
        threadBButton->setText(tr("Stop B"));
    }
}
```

startOrStopThreadB()의 코드는 아주 간단하다.

```
void ThreadForm::closeEvent(QCloseEvent *event)
{
    threadA.stop();
    threadB.stop();
    threadA.wait();
    threadB.wait();
    event->accept();
}
```

사용자가 Quit를 찰칵하거나 창문을 닫으면 실행중에 있는 스레드들을 정지하고 QCloseEvent::accept()를 호출하기전에 그것들이 끝나기를 기다린다. (QThread::wait()를 사용한다.) 이것은 이 실례에서는 문제가 없다하더라도 응용프로그람이 깨끗한 상태에서 완료하도록

한다.

응용프로그람을 콤파일하려면 .pro파일에 다음 행을 추가해야 한다.

```
CONFIG += thread
```

이것은 qmake가 Qt서고의 스레드판을 사용하게 한다. 스레드화된 Qt서고를 건설하려면 -thread지령행선택을 넘기여 Unix와 Mac OS X에서 스크립트환경을 구성해야 한다. Windows에서 Qt서고는 기정으로 스레드화된다. 또한 Windows의 콘솔에 프로그람의 출력이 나타나게 하려고 하므로 콘솔선택을 요구한다.

```
win32:CONFIG += console
```

응용프로그람을 실행하고 Start A를 찰칵하면 콘솔은 'A'들로 채워진다. Start B를 찰칵하면 'A'와 'B'들이 엇바뀌는 렬들이 출력된다. Stop A를 찰칵하면 오직 'B'들이 출력된다.

다중스레드 응용프로그람들에서 일반적인 요구는 여러개의 스레드들을 동기시키는것이다. Qt는 이것을 수행하는 QMutex, QMutexLocker, QSemaphore 및 QWaitCondition클라스들을 제공한다.

QMutex클라스는 변수 혹은 코드부분을 보호하는 수단을 제공함으로써 오직 하나의 스레드를 한번에 호출할수 있다. 이 클라스는 뮤텍스를 잠그는 lock()함수를 제공한다. 뮤텍스가 잠그어지지 않았으면 현재 스레드는 곧 그것을 포착하고 잠그며 그렇지 않으면 현재스레드는 뮤텍스를 보유하는 스레드가 뮤텍스를 열 때까지 차단된다. 한편 lock()호출이 돌아올 때 현재 스레드는 unlock()를 호출할 때까지 뮤텍스를 보유한다. 또한 QMutex는 뮤텍스가 이미 잠그어져있으면 곧 돌아오는 tryLock()함수를 제공한다.

례를 들면 QMutex를 가지는 Thread클라스의 stopped변수를 보호하려고 한다고 가정하자. 그때 Thread에 다음의 자료성원을 추가한다.

```
QMutex mutex;
```

run()함수는 다음과 같이 변경한다.

```
void Thread::run()
{
    for (;;) {
        mutex.lock();
        if (stopped) {
            stopped = false;
            mutex.unlock();
            break;
        }
        mutex.unlock();
        cerr << messageStr.ascii();
    }
```

```
        cerr << endl;
    }
```

stop()함수는 다음과 같다.

```
    void Thread::stop()
    {
        mutex.lock();
        stopped = true;
        mutex.unlock();
    }
```

복잡한 함수들 특히 C++례외를 사용하는 함수들에서 뮤텍스의 잠건과 열기는 오유를 일으키기 쉽다. Qt는 QMutexLocker편의클라스를 제공하여 뮤텍스조종을 단순화한다. QMutexLocker의 구성자는 QMutex를 인수로 받아들이고 그것을 잠근다. QMutexLocker의 해체자는 뮤텍스의 잠건을 연다. 례를 들면 우의 stop()함수를 다음과 같이 다시 쓴다.

```
    void Thread::stop()
    {
        QMutexLocker locker(&mutex);
        stopped = true;
    }
```

QSemaphore는 Qt에서 쎄마퍼를 제공한다. 쎄마퍼는 뮤텍스의 일반화로서 일정한 수의 동일한 자원들을 보호하는데 사용할수 있다.

다음 2개의 코드부분은 QSemaphore와 QMutex사이의 대응관계를 보여준다.

```
QSemaphore semaphore(1);          QMutex mutex;
semaphore++;                      mutex.lock();
semaphore--;                      mutex.unlock();
```

뒤붙이식 ++와 --연산자들은 쎄마퍼가 보호하는 하나의 자원을 얻어서 넘겨준다. 구성자에 1을 넘기여 쎄마퍼에 신호자원을 조종하라고 알린다. 쎄마퍼사용의 우점은 구성자에 1을 넘기고 ++를 여러번 호출하여 많은 자원을 얻을수 있는것이다.

쎄마퍼의 전형적인 응용은 2개 스레드들사이에서 일정한 량의 자료(DataSize)를 일정한 크기(BufferSize)의 공유원형완충기를 리용하여 전송하는 경우이다.

```
const int DataSize = 100000;
const int BufferSize = 4096;
char buffer[BufferSize];
```

생산자스레드는 완충기의 끝에 이를 때까지 자료를 완충기에 써넣고 선두로부터 시작하여 현존자료를 덧쓰기한다. 소비자스레드는 생성될 때 자료를 읽어들인다. 그림 17-2에서는 아주 작은 16byte완충기를 가정하고 이것을 설명한다.

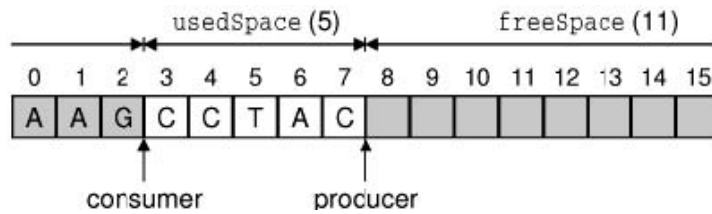



그림 17-2. 생산자-소비자모형

생산자-소비자실례에서는 동기화의 필요성이 두가지 있다. 즉 생산자가 자료를 너무 빨리 생성하면 소비자가 아직 읽어들이지 못한 자료를 덧쓰기한다. 소비자가 자료를 너무 빨리 읽어들이면 생산자가 아직 채 쓰지 못한 부분까지도 읽어들인다.

이 문제를 해결하는 원시적인 수법은 생산자가 완충기를 채우게 하고 소비자가 전체 완충기를 읽어들일 때까지 기다리는것이다. 그러나 다중처리소자콤퓨터들에서 이것은 생산자와 소비자스레드들이 완충기의 서로 다른 부분들에 대하여 동시에 조작할만큼 빠르지 못하다.

문제를 효과적으로 해결하는 한가지 수법은 2개의 쎄마퍼를 사용하는것이다.

```
QSemaphore  freeSpace(BufferSize);
QSemaphore  usedSpace(BufferSize);
```

freeSpace쎄마퍼는 생산자가 자료를 채울수 있는 완충기부분을 관리한다. usedSpace쎄마퍼는 소비자가 읽어들일수 있는 령역을 관리한다. 이 2개 령역은 보충적이다. 둘다 BufferSize(4096)로 초기화되며 이것은 그렇게 많은 자원을 관리할수 있다는것을 의미한다.

이 실례에서 매 바이트는 하나의 자원으로 계수된다. 현실세계의 응용프로그람에서는 더 큰 단위(례를 들면 한번에 64 혹은 256byte)에 대하여 조작하여 쎄마퍼사용으로 인한 추가적인 부담을 줄이게 한다.

```
void  acquire(QSemaphore  &semaphore)
{
    semaphore++;
}
```

acquire()함수는 하나의 자원(완충기의 1byte)을 얻으려고 한다. QSemaphore는 여기에 뒤붙이식 ++연산자를 사용하지만 우리의 실례에서는 acquire()라는 함수를 사용하는것이 더 객관적이다.

```
void  release(QSemaphore  &semaphore)
{
    semaphore--;
}
```

마찬가지로 release()함수를 뒤붙이식 --연산자의 동의어로 실현한다.

```
void  Producer::run()
{
```

```
    for (int i = 0;i < DataSize;++i) {
        acquire(freeSpace);
        buffer[i % BufferSize] = "ACGT"[ (uint)rand() % 4];
        release(usedSpace);
    }
}
```

생산자에서는 하나의 《자유》바이트를 얻는것으로 시작한다. 완충기가 소비자가 아직 읽지 못한 자료로 꽉 차면 acquire()호출은 소비자가 자료를 소비하기 시작할 때까지 기다린다. 일단 그 바이트를 얻었다면 그것을 우연자료('A', 'C', 'G', 혹은 'T')로 채우고 그 바이트를《사용된》 바이트로 풀어놓음으로써 그것을 소비자스레드가 읽을수 있게 한다.

```
void Consumer::run()
{
    for (int i = 0; i < DataSize; ++i) {
        acquire(usedSpace);
        cerr << buffer[i % BufferSize];
        release(freeSpace);
    }
    cerr << endl;
}
```

소비자에서는 《사용된》 바이트를 얻는것으로 시작한다. 완충기에 읽어들일 자료가 더는 없으면 acquire()호출은 생산자가 생성할 때까지 기다린다. 바이트를 얻었다면 그것을 출력하고 바이트를 《자유》바이트로서 풀어놓아 생산자가 다시 자료를 채울수 있게 한다.

```
int main()
{
    usedSpace += BufferSize;
    Producer producer;
    Consumer consumer;
    producer.start();
    consumer.start();
    producer.wait();
    consumer.wait();
    return 0;
}
```

끝으로 main()에서 QSemaphore의 비직관적인 +=연산자를 리용하여 《사용된》 공간을 모두 얻는것으로 시작한다. 이것은 채 쓰지 않은 부분을 소비자가 읽어들이지 않도록 한다. 그

다음 생산자와 소비자스레드들을 기동한다. 그때 생기는 현상은 생산자가 《자유》공간을 《사용된》공간으로 변환하고 그다음 소비자는 그것을 《자유》공간으로 역변환할수 있다는것이다.

프로그람을 실행할 때 콘솔에 100000개의 'A', 'C', 'G', 'T'들의 우연렬을 써넣고 완료한다. 그때 수행하는 작업을 리해하기 위하여 출력장치에 써넣기를 금지하고 그대신에 생산자가 바이트를 생성할 때마다 'P'를 쓰고 소비자가 바이트를 읽을 때마다 'c'를 써넣는다. 그리고 될수록 간단히 모든 동작이 진행되도록 하기 위하여 DataSize와 BufferSize에 훨씬 더 작은 값을 사용할수 있다.

례를 들면 여기에 DataSize가 10, BufferSize가 4일 때 가능한 실행이 있다. 즉 "PcPcPcPcPcPcPcPcPcPc"이다. 이 경우에 소비자는 생산자가 바이트들을 생성할 때마다 곧 읽어들이며 2개의 스레드는 같은 속도로 실행된다. 다른 가능성은 소비자가 완충기를 읽기 시작하기전에 생산자가 전체 완충기를 채우는것이다. 즉 "PPPPccccPPPPccccPPcc"이다. 다른 가능성도 많다. 쎄마퍼는 체계에 고유한 스레드일정계획기에 많은 자유범위를 준다. 스레드일정계획기는 스레드들의 동작을 탐구하여 최적인 일정계획방략을 선택할수 있다.

생산자와 소비자를 동기시키는 문제를 해결하는 다른 수법은 QWaitCondition과 QMutex를 사용하는것이다. QWaitCondition은 어떤 조건이 만족될 때 한 스레드가 다른 스레드들을 깨우게 한다. 이것은 오직 뮤텍스들에 의해서만 가능한것보다 더 정확한 조종을 허용한다. 그 동작을 보여주기 위하여 기다림조건들을 사용하여 생산자-소비자실례를 다시 시행한다.

```
const int DataSize = 100000;
const int BufferSize = 4096;
char buffer[BufferSize];
QWaitCondition bufferIsNotFull;
QWaitCondition bufferIsNotEmpty;
QMutex mutex;
int usedSpace = 0;
```

완충기외에도 2개의 QWaitCondition, 한개의 QMutex, 그리고 완충기안에 《사용된》 바이트가 몇개인가를 보관하는 변수를 하나 선언한다.

```
void Producer::run()
{
    for (int i = 0;i < DataSize;++i) {
        mutex.lock();
        while (usedSpace == BufferSize)
            bufferIsNotFull.wait(&mutex);
        buffer[i % BufferSize] = "ACGT"[ (uint)rand() % 4];
        ++usedSpace;
        bufferIsNotEmpty.wakeAll();
```

```
            mutex.unlock();
        }
    }
```

생산자에서는 완충기가 찼는가 검사하는것으로 시작한다. 다 찼으면 《완충기가 차지 않음》 조건이 성립하기를 기다린다. 조건이 만족되면 완충기에 1byte 써넣고 usedSpace를 증가시키고 《완충기가 비지 않음》 조건이 true로 되기를 기다리는 스레드를 잠재운다.

뮤텍스를 사용하여 usedSpace변수에 대한 모든 호출을 보호한다. QWaitCondition::wait()함수는 잠견된 뮤텍스를 첫 인수로 가지며 현재 스레드를 차단하기전에 뮤텍스를 돌려준다.

이 실례에서는 while순환

```
        while (usedSpace == BufferSize)
            bufferIsNotFull.wait(&mutex);
```

을 if문으로 교체한다.

```
        if (usedSpace == BufferSize) {
            mutex.unlock();
            bufferIsNotFull.wait();
            mutex.lock();
        }
```

그러나 이것은 하나이상의 생산자스레드를 허용하자마자 차단된다. 그것은 다른 생산자가 wait()호출후 곧 뮤텍스를 동결하고 《완충기가 차지 않음》 조건을 다시 false로 만들수 있기때문이다.

```
    void Consumer::run()
    {
        for (int i = 0;i < DataSize;++i) {
            mutex.lock();
            while (usedSpace == 0)
                bufferIsNotEmpty.wait(&mutex);
            cerr << buffer[i % BufferSize];
            --usedSpace;
            bufferIsNotFull.wakeAll();
            mutex.unlock();
        }
        cerr << endl;
    }
```

소비자는 생산자와 정반대로 작업한다. 즉 소비자는 《완충기가 비지 않음》조건을 기다리며 《완충기가 차지 않음》 조건을 기다리는 스레드를 깨운다.

지금까지의 모든 실례에서 스레드들은 같은 대역변수들을 호출하였다. 그러나 일부 스레드응용프로그람들은 서로 다른 스레드들에서 각이한 값을 보유하는 대역변수를 가질 필요가 있다. 이것은 흔히 스레드국부기억(thread-local storage, TLS) 혹은 스레드고유기억(thread-specific data, TSD)이라고 부른다. QThread::currentThread()로부터 돌아오는 스레드ID들을 건으로 하는 매프를 리용하여 그것을 꾸밀수 있지만 더 좋은 수법은 QThreadStorage<T>클라스를 사용하는 것이다.

QThreadStorage<T>의 일반적인 용도는 고속완충기억기이다. 서로 다른 클라스들에서 제각기 캐쉬를 취함으로써 뮤텍스에서 잠그기, 열기가 가능한 기다림으로 인한 추가적인 부담을 피할수 있다. 례를 들면

```
QThreadStorage<QMap<int, double> *> cache;
void insertIntoCache(int id, double value)
{
    if (!cache.hasLocalData())
        cache.setLocalData(new QMap<int, double>);cache.localData()->insert(id, value);
}
void removeFromCache(int id)
{
    if (cache.hasLocalData())
        cache.localData()->remove(id);
}
```

cache변수는 스레드마다 QMap<int, double>의 지적자를 하나 가진다. (일부 콤파일러들에서 문제가 있으므로 QThreadStorage<T>에서 형판형은 지적자형이여야 한다.) 특정한 스레드에서 캐쉬를 처음 리용할 때 hasLocalData()는 false를 돌려주고 QMap<int, double>객체를 창조한다.

캐쉥과 함께 QThreadStorage<T>는 대역오유상태변수(toerrno와 비슷하다)에 쓰이며 한개 스레드에서의 수정이 다른 스레드들에 영향을 주지 않도록 한다.

제2절. GUI스레드와의 교제

Qt응용프로그람이 기동할 때 오직 하나의 스레드 즉 초기스레드가 실행중에 있다. 이것은 QApplication객체를 창조하고 그것에 대하여 exec()를 호출하게 하는 유일한 스레드이다. 이러한 리유로 보통 이 스레드를 GUI스레드로 취급한다. exec()호출후에 이 스레드는 사건을 기다리거나 사건을 처리한다.

GUI스레드는 앞 절에서 수행한것처럼 QThread파생클라스의 객체를 창조하여 새로운 스레드들을 기동할수 있다. 새로운 스레드들은 서로 교제할 필요가 있으면 뮤텍스, 쎄마퍼 혹은 기다림조건들과 함께 공유변수들을 사용할수 있다. 그러나 이 기술은 사건순환고리를 잠그고 사용자대면부를 동결할수 있으므로 GUI스레드와 교제하는데 사용할수 없다.

비GUI스레드가 GUI스레드와 교제하기 위한 대책은 사용자정의사건들을 사용하는것이다.

Qt의 사건기구는 기본형과 함께 사용자정의사건형들을 정의하고 QApplication::postEvent()를 리용하여 이 형의 사건들을 발송할수 있다. 더우기 postEvent()가 스레드에 안전하므로 임의의 스레드로부터 이 함수를 리용하여 GUI스레드에로 사건을 발송할수 있다.

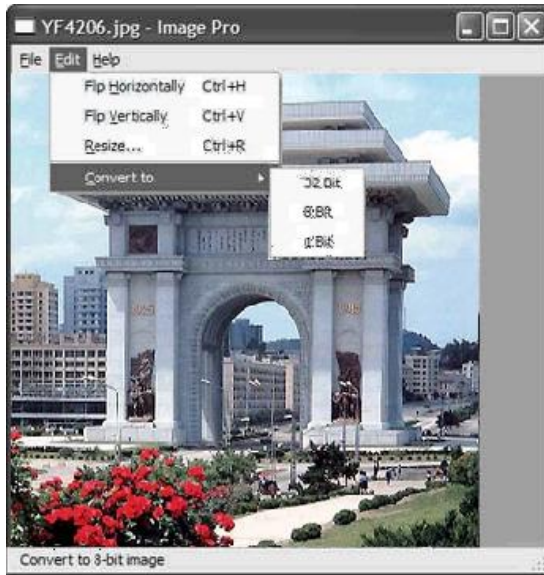



그림 17-3. Image Pro응용프로그람

그 동작을 설명하기 위하여 사용자가 화상의 회전, 크기조절, 색깊이변경을 가능하게 하는 기본화상처리프로그람인 Image Pro의 코드를 고찰한다. 응용프로그람은 비GUI스레드를 리용하여 사건순환고리를 잠그지 않고 화상에 대한 조작을 수행한다. 이것은 큰 화상을 처리할 때 크게 차이난다. 비GUI스레드는 수행해야 할 과제 또는 《일괄처리》들의 목록을 가지며 사건들을 기본창문에 보내여 진척상황을 알린다.

```
ImageWindow::ImageWindow(QWidget *parent, const char *name)
    : QMainWindow(parent, name)
{
    thread.setTargetWidget(this);
    ...
}
```

ImageWindow구성자에서는 비GUI스레드의 《목표창문부품》를 ImageWindow로 설정한다. 이 스레드는 그 창문부품에 진척상황사건들을 발송한다. 스레드변수는 TransactionThread형이다. 그것을 간단히 설명한다.

```
void ImageWindow::flipHorizontally()
{
    addTransaction(new FlipTransaction(Horizontal));
}
```

flipHorizontally()처리부는 FlipTransaction을 창조하고 비공개함수 addTransaction()을 사용하

여 그것을 등록한다. flipVertical(), resizeImage(), convertTo32Bit(), convertTo8Bit(), convertTo1Bit() 함수들도 비슷하다.

```
void ImageWindow::addTransaction(Transaction *transact)
{
    thread.addTransaction(transact);
    openAct->setEnabled(false);
    saveAct->setEnabled(false);
    saveAsAct->setEnabled(false);
}
```

addTransaction()함수는 일괄처리를 비GUI스레드의 일괄처리기다림렬에 추가하고 일괄처리들을 처리하는동안 Open, Save, Save As작용을 금지한다.

```
void ImageWindow::customEvent(QCustomEvent *event)
{
    if ((int)event->type() == TransactionStart) {
        TransactionStartEvent *startEvent = (TransactionStartEvent *)event;
        infoLabel->setText(startEvent->message);
    } else if ((int)event->type() == AllTransactionsDone) {
        openAct->setEnabled(true);
        saveAct->setEnabled(true);
        saveAsAct->setEnabled(true);
        imageLabel->setPixmap(QPixmap(thread.image()));
        infoLabel->setText(tr("Ready"));
        modLabel->setText(tr("MOD"));
        modified = true;
        statusBar()->message(tr("Done"), 2000);
    } else {
        QMainWindow::customEvent(event);
    }
}
```

customEvent()함수는 사용자정의사건들을 처리할수 있게 QObject로부터 재정의된다. TransactionStart와 AllTransactionsDone상수들은 transactionthread.h에서 다음과 같이 정의된다.

```
enum { TransactionStart = 1001, AllTransactionsDone = 1002 };
```

Qt의 기본사건들은 1000아래의 값을 가진다. 더 큰 값들은 사용자정의사건들에 사용될수 있다.

사용자정의사건들의 자료형은 사건형과 함께 void지적자를 보관하는 QEvent의 파생클라

스 QCustomEvent이다. TransactionStart사건들에서는 추가자료성원을 보관하는 QCustomEvent의 파생클라스를 사용한다.

```
class TransactionStartEvent : public QCustomEvent
{
public:
    TransactionStartEvent();
    QString message;
};
TransactionStartEvent::TransactionStartEvent() : QCustomEvent(TransactionStart) {}
```

구성자에서는 TransactionStart상수를 기초클라스구성자에 넘긴다.

그러면 TransactionThread클라스를 고찰하자.

```
class TransactionThread : public QThread
{
public:
    void run();
    void setTargetWidget(QWidget *widget);
    void addTransaction(Transaction *transact);
    void setImage(const QImage &image);
    QImage image();
private:
    QWidget *targetWidget;
    QMutex mutex;
    QImage currentImage;
    std::list<Transaction *> transactions;
};
```

TransactionThread클라스는 배경에서 일괄처리들을 하나씩 처리하고 실행하기 위하여 일괄처리의 목록을 유지관리한다.

```
void TransactionThread::addTransaction(Transaction *transact)
{
    QMutexLocker locker(&mutex);
    transactions.push_back(transact);
    if (!running())
        start();
}
```

addTransaction()함수는 일괄처리기다림렬에 일괄처리를 추가하고 그것이 이미 실행중에 있

지 않으면 일괄처리스레드를 기동한다.

```
void TransactionThread::run()
{
    Transaction *transact;
    for ( ; ; ) {
        mutex.lock();
        if (transactions.empty()) {
            mutex.unlock();
            break;
        }
        QImage oldImage = currentImage;
        transact = *transactions.begin();
        transactions.pop_front();
        mutex.unlock();
        TransactionStartEvent *event = new TransactionStartEvent;
        event->message = transact->messageStr();
        QApplication::postEvent(targetWidget, event);
        QImage newImage = transact->apply(oldImage);
        delete transact;
        mutex.lock();
        currentImage = newImage;
        mutex.unlock();
    }
    QApplication::postEvent(targetWidget, new QCustomEvent(AllTransactionsDone));
}
```

run()함수는 일괄처리기다림렬을 순환하면서 apply()를 호출하여 매개 일괄처리를 차례로 실행한다. 일괄처리들과 currentImage성원변수에 대한 모든 호출은 뮤텍스에 의해 보호된다.

일괄처리가 기동할 때 TransactionStart사건을 목표창문부품(ImageWindow)에 발송한다. 모든 일괄처리가 처리를 완료하였을 때 AllTransactionsDone사건을 발생한다.

```
class Transaction
{
public:
    virtual QImage apply(const QImage &image) = 0;
    virtual QString messageStr() = 0;
};
```

Transaction클라스는 사용자가 화상에 대하여 조작하는 추상기초클라스이다. 이것은 3개의 파생클라스 FlipTransaction, ResizeTransaction, ConvertDepthTransaction을 가진다. FlipTransaction만 고찰하는데 다른 2개 클라스들은 비슷하다.

```
class FlipTransaction : public Transaction
{
public:
    FlipTransaction(Qt::Orientation orient);
    QImage apply(const QImage &image);
    QString messageStr();
private:
    Qt::Orientation orientation;
};
```

FlipTransaction구성자는 방향(Horizontal 혹은 Vertical)을 지정하는 하나의 파라메터를 가진다.

```
QImage FlipTransaction::apply(const QImage &image)
{
    return image.mirror(orientation == Qt::Horizontal, orientation == Qt::Vertical);
}
```

apply()함수는 파라메터로 받아들이는 QImage에 대하여 QImage::mirror()를 호출하고 결과 QImage를 돌려준다.

```
QString FlipTransaction::messageStr()
{
    if (orientation == Qt::Horizontal)
       return QObject::tr("Flipping image horizontally...");
    else
       return QObject::tr("Flipping image vertically...");
}
```

messageStr()는 조작이 진행되는 동안 상태띠에 표시할 통보문을 돌려준다. 이 함수는 ImageWindow::customEvent()와 GUI스레드에서 호출된다.

실행이 오래 걸리는 조작들에서는 진척상황을 알릴 필요가 있다. 추가적인 사용자정의사건을 창조하고 일정한 퍼센트의 처리가 끝나면 그것을 발송함으로써 수행할수 있다.

제3절. 비GUI스레드들에서 Qt클라스들의 사용

함수는 서로 다른 스레드들로부터 동시에 안전하게 호출할수 있을 때 스레드안전함수라고 말한다. 2개의 스레드안전함수를 서로 다른 스레드들로부터 같은 공유자료에 대하여 호출한다면 결과가 늘 정의된다. 더 확장하여 클라스는 그의 모든 함수들이 서로 다른 스레드들

로부터 지어는 같은 객체에 대하여 조작할 때에도 서로 간섭하지 않고 동시에 호출할수 있을 때 스레드안전클라스라고 말한다.

Qt의 스레드안전클라스들은 QThread, QMutex, QMutexLocker, QSemaphore, QThreadStorage <T> 및 QWaitCondition이다. 또한 다음의 함수들은 스레드안전함수이다. 즉 QApplication::postEvent(), QApplication::removePostedEvents(), QApplication::removePostedEvent() 및 QEventLoop::wakeUp().

Qt의 대다수 비GUI클라스들은 좀 엄격한 요구를 만족시켜야 한다. 그것은 재입구가능(*reentrant*)이다. 클라스의 각이한 실례들을 각이한 스레드들에서 동시에 사용할수 있으면 그 클라스는 재입구가능이다. 그러나 여러 스레드들사이에서 동시에 같은 재입구가능객체를 호출하는것은 안전하지 못하고 그러한 호출은 뮤텍스에 의해 보호되여야 한다. 재입구가능클라스들은 Qt참고문서에서와 같이 표식된다. 일반적으로 대역적으로 참고하지 않는 임의의 C++ 클라스 혹은 공유자료는 재입구가능이다.

QObject는 재입구가능이지만 Qt의 어느 QObject파생클라스도 재입구가능이 아니다. 이로부터 얻어지는 하나의 결론은 비GUI스레드로부터 창문부품에 대하여 함수들을 직접 호출할수 없다는것이다. 비GUI스레드로부터 QLabel의 본문을 변경하려고 한다면 사용자정의사건을 GUI스레드에 발송하여 본문을 변경할것을 요구해야 한다.

delete에 의한 QObject삭제는 재입구가능이 아니다. 비GUI스레드로부터 QObject를 삭제하기 위하여 《기한부삭제》사건을 발송하는 QObject::deleteLater()을 호출해야 한다.

QObject의 신호-처리부기구는 임의의 스레드에서 사용될수 있다. 신호가 한개 스레드에서 발생될 때 거기에 련결된 처리부들은 즉시 호출되고 실행은 같은 스레드에서 발생하고 수신자객체가 창조되는 스레드에서는 발생하지 않는다. 이것은 신호와 처리부를 사용하여 다른 스레드들로부터 GUI스레드와 교제할수 없다는것을 의미한다.

QTimer클라스와 망프로그람작성클라스 QFtp, QHttp, QSocket, 및 QSocketNotifier는 모두 사건순환고리에 의존하므로 비GUI스레드에서 사용할수 없다. 유효한 단 하나의 망구축클라스는 가동환경에 고유한 망구축API들을 위한 저수준래퍼로서 QSocketDevice이다. 일반적인 기술은 비GUI스레드에서 동기QSocketDevice를 사용하는것이다. 일부 프로그람작성자들은 QSocket(비동기적으로 작업한다.)를 사용하는 경우보다 코드를 더 단순하게 한다는것을 발견하고 비GUI 스레드에서 작업함으로써 사건순환고리를 차단하지 않는다.

또한 Qt의 SQL과 OpenGL모듈은 다중스레드응용프로그람들에서 쓰일수 있으나 체계에 따라 변하는 자체의 제한을 가지고있다.

QImage, QString, 용기클라스들을 비롯한 Qt의 대다수 비GUI클라스들은 최적화기술로서 암시적 혹은 명시적공유를 사용한다. 이 클라스들은 복사구성자와 대입연산자들을 제외하고 재입구가능이다. 이러한 클라스들의 실례의 사본을 취할 때 오직 내부자료의 지적자가 복사된다. 이것은 여러개의 스레드들이 자료를 동시에 수정하려고 하는 경우에는 위험하다. 그러한 경우에 해결책은 암시적 혹은 명시적으로 공유된 클라스의 실례에 대입을 할 때

QDeepCopy<T>클라스를 사용하는것이다. 례를 들면

```
QString password;
QMutex mutex;
void setPassword(const QString &str)
{
   mutex.lock();
   password = QDeepCopy<QString>(str);
   mutex.unlock();
}
```

제18장. 가동환경에 고유한 기능

이 장에서는 Qt프로그람작성자에게 유효한 가동환경에 고유한 일부 선택을 고찰한다. 우선 Windows에서 Win32 API, Mac OS X에서 Core Graphics, X11에서 Xlib와 같은 API들을 호출하는 방법을 고찰한다. 그다음 Qt의 ActiveQt확장을 조사하고 Qt/Windows응용프로그람들에서 ActiveX조종요소들의 사용법과 ActiveX봉사기들에서 동작하는 응용프로그람들을 창조하는 방법을 보여준다. 그리고 마지막 절에서는 X11하에서 Qt응용프로그람들이 쎄숀관리기와 협동하는 방법을 설명한다.

여기서 제시하는 기능들과 함께 Qt Enterprise판에는 Motif와 Xt응용프로그람들을 Qt에 간단히 옮기게 하는 Qt/Motif확장이 포함되여있다. Tcl/Tk응용프로그람용의 비슷한 확장을 froglogic가 제공하며 Microsoft Windows자원변환기를 Klarälvdalens Datakonsult로부터 사용할수 있다. 그리고 매몰판개발을 위하여 Trolltech는 Qtopia응용프로그람틀거리를 제공한다.

제1절. 원시적인 API들과의 대면

Qt는 모든 가동환경에서 대부분의 요구를 만족시키는 포괄적인 API들을 제공한다. 그러나 일부 환경에서는 기초하고있는 가동환경에 고유한 API들을 사용하려고 할수 있다. 이 절에서는 특별한 과제를 수행하기 위하여 Qt가 제공하는 각이한 가동환경들에서 원시적인 API들을 사용하는 방법을 보여준다.

매개 가동환경에서 QWidget는 창문ID(Windows에서 HWND)를 돌려주는 winId()함수를 제공한다. 또한 QWidget는 특별한 창문ID를 가지는 QWidget를 돌려주는 find()라는 정적함수를 제공한다. 이 ID를 원시API함수들에 넘기여 가동환경에 고유한 효과를 얻는다. 례를 들면 다음의 코드는 winId()를 사용하여 원시Core Graphics함수들(Qt 3.3은 원시API호출에 의거하지 않고도 이것을 달성하는 함수를 제공한다.)을 리용하여 Mac OS X에서 QLabel을 반투명으로 만든다.

```
#include <qapplication.h>
#include <qlabel.h>
#include <qt_mac.h>
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    QLabel *label = new QLabel("Hello Qt!", 0);
    app.setMainWidget(label);
    CGSWindowRef winRef = GetNativeWindowFromWindowRef((WindowRef)label->winId());
    CGSSetWindowAlpha(_CGSDefaultConnection(), winRef, 0.5);
    label->show();
```

```
    return app.exec();
}
```

여기에 Win32 API를 사용하여 Windows에서 같은 효과를 얻는 방법이 있다.

```
#define _WIN32_WINNT 0x0501
#include <qapplication.h>
#include <qt_windows.h>
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    QLabel *label = new QLabel("Hello Qt!", 0);
    app.setMainWidget(label);
    int exstyle = GetWindowLong(label->winId(), GWL_EXSTYLE);
    exstyle |= WS_EX_LAYERED;
    SetWindowLong(label->winId(), GWL_EXSTYLE, exstyle);
    SetLayeredWindowAttributes(label->winId(), 0, 128, LWA_ALPHA);
    Label->show();
    return app.exec();
}
```

이 코드는 가동환경이 Windows 2000 혹은 XP라는것을 전제로 한다. 반투명기능을 유지하지 않는 낡은 판의 Windows에서 응용프로그람을 콤파일하고 실행하려고 한다면 QLibrary를 사용하여 련결시가 아니라 실행시에 SetLayeredWindowAttributes기호를 해결할수 있다.

```
typedef BOOL (__stdcall *PSetLayeredWindowAttributes)
                                    (HWND, COLORREF, BYTE, DWORD);
PSetLayeredWindowAttributes pSetLayeredWindowAttributes =
      (PSetLayeredWindowAttributes) QLibrary::resolve("user32", "SetLayeredWindowAttributes");
if (pSetLayeredWindowAttributes) {
     int exstyle = GetWindowLong(label->winId(), GWL_EXSTYLE);
     exstyle |= WS_EX_LAYERED;
     SetWindowLong(label->winId(), GWL_EXSTYLE, exstyle);
     pSetLayeredWindowAttributes(label->winId(), 0, 128, LWA_ALPHA);
}
```

Qt/Windows는 이 기술을 내적으로 사용하여 Qt응용프로그람들이 본래의 유니코드유지와 효과적인 서체변환과 같은 고급한 기능의 우점들을 리용하며 여전히 Windows의 낡은 판들에서 실행할수 있도록 한다.

X11에서 투명성을 얻는 표준방법은 없다. 그러나 여기에 X11창문속성을 수정하는 방법이

있다.

```
Atom atom = XInternAtom(win->x11Display(), "MY_PROPERTY", False);
long data = 1;
XChangeProperty(win->x11Display(), win->winId(), atom, atom, 32, PropModeReplace,
                (unsigned char *)&data, 1);
```

Qt/Embedded는 원시API없이 Linux틀완충기에 직접 실현하는 Qt판들과 다르다. 또한 자체의 창문체계 QWS를 제공한다. 이것은 QWSDecoration과 QWSInputMethod와 같은 Qt/Embedded에 고유한 클라스들의 파생클라스를 만듬으로써 구성될수 있다. Qt/Embedded의 다른 한가지 차이는 사용하지 않은 클라스들과 기능들을 콤파일하지 않음으로써 크기를 줄일수 있다는것이다.

다른 이식가능한 Qt응용프로그람에서 가동환경에 고유한 코드를 사용하려고 한다면 원시코드를 #if와 #endif안에 넣을수 있다. 례를 들면

```
#if defined(Q_WS_MAC)
   CGSWindowRef winRef = GetNativeWindowFromWindowRef((WindowRef)label->winId());
   CGSSetWindowAlpha(_CGSDefaultConnection(), winRef, 0.5);
#endif
```

Qt는 다음 4가지 창문체계기호 즉 Q_WS_WIN, Q_WS_X11, Q_WS_MAC 그리고 Q_WS_QWS를 정의한다. 응용프로그람들에서 이 기호들을 사용하기전에 적어도 하나의 Qt머리부에서 포함한다는것을 확인해야 한다. 또한 Qt는 조작체계를 식별하기 위한 앞처리기기호들을 제공한다. 즉

Q_OS_WIN32	Q_OS_DGUX	Q_OS_LINUX	Q_OS_QNX6
Q_OS_WIN64	Q_OS_DYNIX	Q_OS_LYNX	Q_OS_RELIANT
Q_OS_CYGWIN	Q_OS_FREEBSD	Q_OS_NETBSD	Q_OS_SCO
Q_OS_MAC	Q_OS_HPUX	Q_OS_OPENBSD	Q_OS_SOLARIS
Q_OS_AIX	Q_OS_HURD	Q_OS_OSF	Q_OS_ULTRIX
Q_OS_BSDI	Q_OS_IRIX	Q_OS_QNX	Q_OS_UNIXWARE

대체로 이 기호들중 하나가 정의된것으로 가정할수 있다. 편리상 Qt는 Win32나 Win64가 탐지되면 Q_OS_WIN을 정의하고 Unix기반조작체계(Mac OS X포함)가 탐지되면 Q_OS_UNIX를 정의한다. 실행시에 QApplication::winVersion() 혹은 QApplication::macVersion()을 호출하여 각이한 판의 Windows (95, 98, 등) 혹은 Mac OS X (10.0, 10.1, 등)을 구별할수 있다.

Qt의 일부 GUI관련클라스들은 객체에로의 저수준핸들을 돌려주는 가동환경에 고유한 handle()함수를 제공한다. 표 18-1은 각이한 가동환경에서 handle()의 돌림값형을 보여준다.

표 18-1. 가동환경에 고유한 핸들형

	Windows	X11	Mac OS X	Embedded
QCursor	HCURSOR	Cursor	int	int

	Windows	X11	Mac OS X	Embedded
QFont	HFONT	Font	FMFontFamily	FontID
QPaintDevice	HDC	Drawable	GWorldPtr	-
QPainter	HDC	Drawable	GWorldPtr	-
QRegion	HRGN	Region	RgnHandle	void *
QSessionManager	-	SmcConn	-	-

QWidget, QPixmap, QPrinter 및 QPicture클라스들은 모두 QPaintDevice를 계승한다. X11과 Mac OS X에서 handle()은 QWidget에서 winId()와 같은것을 의미한다. Windows에서 handle()은 장치상황을 돌려주며 winId()는 창문핸들을 돌려준다. 마찬가지로 QPixmap는 Windows에서 비트매프핸들(HBITMAP)을 돌려주는 hbm()함수를 제공한다.

X11에서 QPaintDevice는 여러가지 지적자들이나 핸들을 돌려주는 많은 함수들(x11Display()와 x11Screen() 등)을 제공한다. 례를 들면 이 함수들을 사용하여 QWidget 혹은 QPixmap에 대하여 X11그라픽스상황을 설정한다.

다른 도구묶음들이나 서고들과 자주 대면해야 하는 Qt응용프로그람들은 저수준사건들(X11에서 XEvent들, Windows와 Mac OS X에서 MSG들, Qt/Embedded에서 QWSEvent들)이 QEvent들로 변환되기전에 호출할 필요가 있다. QApplication의 파생클라스를 만들고 가동환경에 고유한 관련사건려과기(winEventFilter(), x11EventFilter(), macEventFilter(), 및 qwsEventFilter() 중의 하나)를 재정의하여 이것을 수행할수 있다.

winEvent(), x11Event(), macEvent() 그리고 qwsEvent()중의 하나를 재정의함으로써 주어진 QWidget에 송신되는 가동환경에 고유한 사건들을 호출할수 있다. 이것은 오락조종기구의 사건들과 같이 보통 Qt가 무시하는 일정한 형의 사건들을 조종하여 사용할수 있다.

제2절. ActiveX의 사용

Microsoft의 ActiveX기술은 응용프로그람들이 다른 응용프로그람이나 서고들에 의하여 제공되는 사용자대면부 부분품들을 결합하게 한다. 이것은 Microsoft COM우에 구축되고 부분품들을 사용하는 응용프로그람들의 대면부모임과 부분품들을 제공하는 응용프로그람과 서고들의 대면부모임을 정의한다.

Qt/Windows Enterprise판은 ActiveX와 Qt를 원만히 결합하는 ActiveQt틀거리를 제공한다. ActiveQt는 2개의 모듈로 이루어진다. 즉

· *QAxContainer*모듈은 Qt응용프로그람들에서 COM객체들을 사용하게 하고 ActiveX조종요소들을 매몰하게 한다.

· *QAxServer*모듈은 Qt로 쓴 사용자정의COM객체들과 ActiveX조종요소들을 반출하게 한다.

첫 실례는 *QAxContainer*를 리용하여 Qt응용프로그람에 Windows Media Player를 매몰하게 한다. Qt응용프로그람은 Windows Media Player ActiveX조종요소에 *Open*단추, *Play/Pause*단추,

*Stop*단추 그리고 미끄럼띠를 추가한다.

그림 18-1. Media Player응용프로그람

응용프로그람의 기본창문은 PlayerWindow형이다.

```
class PlayerWindow : public QWidget
{
   Q_OBJECT
   Q_ENUMS(ReadyStateConstants)
public:
   enum PlayStateConstants { Stopped = 0, Paused = 1, Playing = 2 };
   enum ReadyStateConstants { Uninitialized=0, Loading=1, Interactive=3, Complete=4 };
   PlayerWindow(QWidget *parent = 0, const char *name = 0);
protected:
   void timerEvent(QTimerEvent *event);
private slots:
   void onPlayStateChange(int oldState, int newState);
   void onReadyStateChange(ReadyStateConstants readyState);
   void onPositionChange(double oldPos, double newPos);
   void sliderValueChanged(int newValue);
   void openFile();
```

PlayerWindow클라스는 QWidget를 계승한다. Q_ENUMS()마크로는 moc에 onReadyStateChange()처리부에서 사용한 ReadyStateConstants형이 렬거형이라는것을 알리는데 필요하다.

```
private:
   QAxWidget *wmp;
   QToolButton *openButton;
   QToolButton *playPauseButton;
```

```
    QToolButton *stopButton;
    QSlider *seekSlider;
    QString fileFilters;
    int updateTimer;
};
```

비공개절에서는 QAxWidget *자료성원을 선언한다.

```
PlayerWindow::PlayerWindow(QWidget *parent, const char *name) : QWidget(parent, name)
{
    ...
    wmp = new QAxWidget(this);
    wmp->setControl("{22D6F312-B0F6-11D0-94AB-0080C74C7E95}");
```

구성자에서는 QAxWidget객체를 창조하여 Windows Media Player ActiveX조종요소를 매몰한다. *QAxContainer*모듈은 3개의 클라스들로 이루어진다. 즉 QAxObject는 COM객체를, QAxWidget는 ActiveX조종요소를 각각 밀봉하며 QAxBase는 QAxObject와 QAxWidget의 핵심 COM기능을 실현한다.

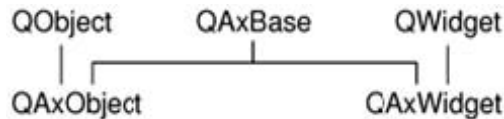



그림 18-2. QAxContainer모듈의 계승나무

Windows Media Player 6.4조종요소의 클라스ID를 가지고 QAxWidget에 대하여 setControl()를 호출한다. 이것은 필요한 부분품의 실례를 창조한다. 그다음 ActiveX조종요소의 모든 속성, 사건 및 메쏘드들을 QAxWidget객체를 통하여 Qt 속성, 신호 및 처리부들처럼 사용할수 있다.

COM자료형들은 표 18-2에 요약한것처럼 대응하는 Qt형들로 자동적으로 변환된다. 례를 들면 VARIANT_BOOL형의 항목파라메터는 bool로, VARIANT_BOOL형의 출구파라메터는 bool &로 된다. 결과형이 Qt클라스(QString, QDateTime 등)이면 항목파라메터는 const참고(례를 들면 const QString &)이다.

표 18-2. COM형과 Qt형들사이의 관계

COM자료형	Qt자료형
VARIANT_BOOL	bool
char, short, int, long	int
unsigned char, unsigned short, unsigned int, unsigned long	uint
float, double	double
CY	Q_LLONG
BST	QString

COM자료형	Qt자료형
DATE	QDateTime
OLE_COLOR	QColor
SAFEARRAY(VARIANT)	QValueList<QVariant>
SAFEARRAY(BYTE)	QByteArray
VARIANT	QVariant
IFontDisp *	QFont
IPictureDisp *	QPixmap

Qt자료형들을 가지는 QAxObject 혹은 QAxWidget에서 사용할수 있는 모든 속성, 신호 및 처리부들의 목록을 얻으려면 generateDocumentation()을 호출하거나 Qt의 extensions\activeqt\example등록부에 배치되여있는 Qt의 dumpdoc지령행도구를 사용해야 한다.

```
wmp->setProperty("ShowControls", QVariant(false, 0));
wmp->setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Expanding);
connect(wmp, SIGNAL(PlayStateChange(int, int)), this, SLOT(onPlayStateChange(int, int)));
connect(wmp, SIGNAL(ReadyStateChange(ReadyStateConstants)), this,
                SLOT(onReadyStateChange(ReadyStateConstants)));
connect(wmp, SIGNAL(PositionChange(double, double)), this,
                SLOT(onPositionChange(double, double)));
```

자체의 단추들을 제공하여 부분품을 조작하므로 PlayerWindow구성자에서는 setControl()호출후에 setProperty()를 호출하여 Windows Media Player의 ShowControls속성을 false로 설정한다. setProperty()함수는 QObject에서 정의되고 COM속성들과 표준Qt속성들에서 모두 사용할수 있다. 함수의 둘째 파라메터는 QVariant형이다. 일부 C++콤파일러들은 아직 bool형속성을 유지하지 않으므로 bool을 가지는 QVariant구성자는 무익한 int파라메터도 가진다. bool외의 형들은 QVariant로 자동변환된다.

다음으로 setSizePolicy()를 호출하여 ActiveX조종요소가 배치관리자의 유효공간을 모두 차지하도록 하고 COM부분품으로부터 3개의 처리부에 3개의 ActiveX사건을 련결한다.

PlayerWindow구성자의 나머지 부분은 보통의 견본을 따르지만 일부 Qt신호를 COM객체가 제공하는 처리부들(Play(),Pause() 그리고 Stop())에 련결하는것만 다르다.

timerEvent()함수를 고찰하자.

```
void PlayerWindow::timerEvent(QTimerEvent *event)
{
    if (event->timerId() == updateTimer) {
        double curPos = wmp->property("CurrentPosition").toDouble();
        onPositionChange(-1, curPos);
    } else {
```

```
        QWidget::timerEvent(event);
    }
}
```

timerEvent()함수는 매체의 부분을 재생하는동안 규칙적인 시격으로 호출된다. 그것을 리용하여 미끄럼띠를 전진시킨다. 이것은 ActiveX조종요소에 대하여 property()를 호출하여 CurrentPosition속성의 값을 QVariant로서 얻고 toDouble()를 호출하여 double로 변환하는 방법으로 수행한다. 그다음 onPositionChange()를 호출하여 갱신한다.

대부분의 코드는 ActiveX와 직접 련관되여있지 않으므로 나머지 코드를 고찰하지 않는다.

*QAxContainer*모듈과 련결하기 위하여서는 .pro파일에 다음의 항목이 필요하다.

```
LIBS += -lqaxcontainer
```

COM객체들을 취급할 때 자주 필요한것은 COM메쏘드를 직접 호출할수 있게 하는것이다. (이것은 Qt신호에 련결하는것과 반대이다.) 이것을 수행하는 가장 쉬운 방법은 첫 파라메터로서 메쏘드의 이름과 서명을, 추가파라메터로서 메쏘드의 인수들을 넘기여 dynamicCall()를 호출하는것이다. 례를 들면

```
wmp->dynamicCall("TitlePlay(uint)", 6);
```

dynamicCall()함수는 QVariant형의 8개 파라메터를 가지며 QVariant를 돌려준다. 이러한 방법으로 IDispatch * 혹은 IUnknown *를 넘기려고 한다면 QAxObject에 부분품을 밀봉하고 그것에 대하여 asVariant()를 호출하여 QVariant로 변환할수 있다. IDispatch * 나 IUnknown * 를 돌려주는 COM메쏘드를 호출하거나 그 형들중의 하나의 COM속성을 호출해야 한다면 그대신 querySubObject()를 사용해야 한다.

```
QAxObject *session = outlook.querySubObject("Session");
QAxObject *defaultContacts =
        session->querySubObject("GetDefaultFolder(OlDefaultFolders)", "olFolderContacts");
```

유지되지 않은 자료형들을 파라메터목록에 넘기여 함수들을 호출하려고 한다면 QAxBase::queryInterface()를 사용하여 COM대면부를 얻고 직접 함수를 호출한다. 대면부를 리용하여 완료했을 때 Release()를 호출해야 한다.

그러한 함수들을 자주 호출해야 한다면 QAxObject나 QAxWidget의 파생클라스를 만들고 COM대면부호출들을 밀봉하는 성원함수들을 제공할수 있다. 그러나 QAxObject와 QAxWidget의 파생클라스들이 자체의 속성, 신호, 처리부들을 정의할수 없다는것을 알아야 한다.

그러면 QAxServer모듈을 고찰하자. 이 모듈은 표준Qt프로그람을 ActiveX봉사기로 절환할수 있게 한다. 봉사기는 공유서고 혹은 독립적인 응용프로그람일수 있다. 공유서고로 구축된 봉사기들은 흔히 프로쎄스내봉사기(in-process server)라고 부르며 독립적인 응용프로그람들은 프로쎄스외봉사기(out-of-process server)라고 부른다.

처음의 QAxServer실례는 프로쎄스내봉사기로서 좌우로 튀는 공을 표시하는 창문부품을 제공한다. 또한 창문부품을 Internet Explorer에 매몰하는 방법을 보여준다.

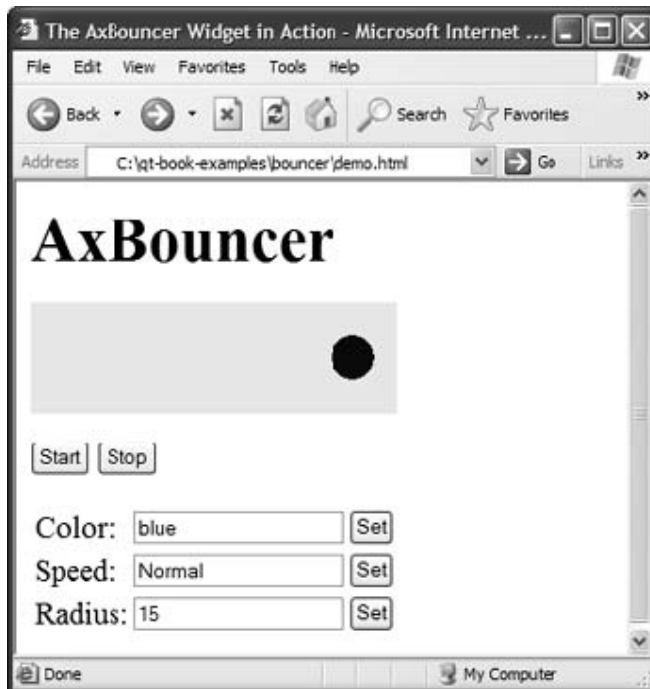



그림 18-3. Internet Explorer에서 AxBouncer창문부품

여기에 AxBouncer창문부품클라스정의의 앞부분이 있다.

```
class AxBouncer : public QWidget, public QAxBindable
{
    Q_OBJECT Q_ENUMS(Speed)
    Q_PROPERTY(QColor color READ color WRITE setColor)
    Q_PROPERTY(Speed speed READ speed WRITE setSpeed)
    Q_PROPERTY(int radius READ radius WRITE setRadius)
    Q_PROPERTY(bool running READ isRunning)
```

AxBouncer는 QWidget와 QAxBindable를 둘다 계승한다. QAxBindable클라스는 창문부품과 ActiveX의뢰기사이의 대면부를 제공한다. 임의의 QWidget를 ActiveX조종요소로서 반출할수 있으나 QAxBindable의 파생클라스를 만들어서 속성값이 변할 때 의뢰기에 통지할수 있으며 COM대면부를 실현하여 QAxServer에 의하여 이미 실현된것들을 보충할수 있다.

QObject의 파생클라스를 계승하는 다중계승을 수행할 때 늘 QObject의 파생클라스를 먼저 놓아서 moc가 그것을 포착하도록 하여야 한다.

3개의 읽고쓰기속성과 하나의 읽기전용속성을 선언한다. Q_ENUMS()마크로는 Speed형이 렬거형이라는것을 moc에게 알리는데 필요하다. Speed렬거는 클라스의 공개부에서 선언된다.

```
public:
    enum Speed { Slow, Normal, Fast };
    AxBouncer(QWidget *parent = 0, const char *name = 0);
```

```
    void setSpeed(Speed newSpeed);
    Speed speed() const { return ballSpeed;}
    void setRadius(int newRadius);
    int radius() const { return ballRadius;}
    void setColor(const QColor &newColor);
    QColor color() const { return ballColor;}
    bool isRunning() const { return myTimerId != 0;}
    QSize sizeHint() const;
    QAxAggregated *createAggregate();
public slots:
    void start();
    void stop();
signals:
    void bouncing();
```

AxBouncer구성자는 parent와 name파라메터를 가지는 창문부품의 표준구성자이다. QAXFACTORY_DEFAULT()마크로는 부분품을 반출하는데 쓰이는데 구성자는 이 서명을 가진다.

createAggregate()함수는 QAxBindable로부터 재정의된다. 그것을 보기로 하자.

```
protected:
    void paintEvent(QPaintEvent *event);
    void timerEvent(QTimerEvent *event);
private:
    int intervalInMilliseconds() const;
    QColor ballColor;
    Speed ballSpeed;
    int ballRadius;
    int myTimerId;
    int x;
    int delta;
};
```

클라스의 보호 및 비공개부는 표준Qt창문부품에서와 같다.

```
AxBouncer::AxBouncer(QWidget *parent, const char *name)
        : QWidget(parent, name, WNoAutoErase)
{
    ballColor = blue;
```

```
        ballSpeed = Normal;
        ballRadius = 15;
        myTimerId = 0;
        x = 20;
        delta = 2;
    }
```

AxBouncer구성자는 클라스의 비공개변수들을 초기화한다.

```
void AxBouncer::setColor(const QColor &newColor)
{
    if (newColor != ballColor && requestPropertyChange("color")) {
        ballColor = newColor;
        update();
        propertyChanged("color");
    }
}
```

setColor()함수는 color속성의 값을 설정한다. 이 함수는 update()를 호출하여 창문부품을 다시 그린다.

류다른 부분은 requestPropertyChange()와 propertyChanged()호출이다. 이 함수들은 QAxBindable을 계승하며 속성을 변경할 때마다 실제로 호출되여야 한다. requestPropertyChange()는 의뢰기의 허가를 받아 속성을 변경하고 의뢰기가 변경을 허용하면 true를 돌려준다. propertyChanged()함수는 의뢰기에 속성이 변경되였다는것을 통지한다.

setSpeed()와 setRadius()속성설정함수들도 이 견본을 따르므로 start()와 stop()처리부들을 실행한다. 이로부터 이 함수들은 실행중에 속성값을 변경한다.

이제는 AxBouncer성원함수를 보기로 하자.

```
QAxAggregated *AxBouncer::createAggregate()
{
    return new ObjectSafetyImpl;
}
```

createAggregate()함수는 QAxBindable로부터 재정의된다. 이 함수는 QAxServer모듈이 이미 실현되지 않은 COM대면부들을 실현하거나 *QAxServer*의 기정COM대면부들을 넘기게 한다. 여기서는 Internet Explorer에서 부분품의 안전한 기능들을 호출하는데 쓰이는 IObjectSafety대면부를 제공한다. 이것은 Internet Explorer의 “Object not safe for scripting”오유통보문을 제거하는 표준기술이다.

여기에 IObjectSafety대면부를 실현하는 클라스의 정의가 있다.

```
class ObjectSafetyImpl : public QAxAggregated, public IObjectSafety
```

```
{
public:
    long queryInterface(const QUuid &iid, void **iface);
    QAXAGG_IUNKNOWN
    HRESULT WINAPI GetInterfaceSafetyOptions(REFIID riid,
            DWORD *pdwSupportedOptions, DWORD *pdwEnabledOptions);
    HRESULT WINAPI SetInterfaceSafetyOptions(REFIID riid,
            DWORD pdwSupportedOptions, DWORD pdwEnabledOptions);
};
```

ObjectSafetyImpl클라스는 QAxAggregated와 IObjectSafety를 모두 계승한다. QAxAggregated 클라스는 추가적인 COM대면부들을 실현하기 위한 추상기초클라스이다. QAxAggregated가 전개하는 COM객체는 controllingUnknown()을 통하여 호출할수 있다. 이 COM객체는 QAxServer 모듈에 의하여 배경에서 창조된다.

QAXAGG_IUNKNOWN마크로는 QueryInterface(), AddRef(), Release()의 표준실현을 제공한다. 이러한 실현은 단지 조종하고있는 COM객체에 대하여 같은 함수들을 호출한다.

```
long ObjectSafetyImpl::queryInterface(const QUuid &iid, void **iface)
{
    *iface = 0;
    if (iid == IID_IObjectSafety)
       *iface = (IObjectSafety *)this;
    else
       return E_NOINTERFACE;
    AddRef();
    return S_OK;
}
```

queryInterface()함수는 QAxAggregated의 순수가상함수이다. 이 함수는 조종하고있는 COM객체에 의하여 호출되여 QAxAggregated의 파생클라스가 제공하는 대면부의 호출을 준다. 실현하지 않은 대면부와 IUnknown에 대하여 E_NOINTERFACE을 돌려주어야 한다.

```
HRESULT WINAPI ObjectSafetyImpl::GetInterfaceSafetyOptions( REFIID riid,
                  DWORD *pdwSupportedOptions, DWORD *pdwEnabledOptions)
{
     *pdwSupportedOptions = INTERFACESAFE_FOR_UNTRUSTED_DATA |
                                   INTERFACESAFE_FOR_UNTRUSTED_CALLER;
     *pdwEnabledOptions = *pdwSupportedOptions;
     return S_OK;
```

```
}
HRESULT WINAPI ObjectSafetyImpl::SetInterfaceSafetyOptions(REFIID, DWORD, DWORD)
        { return S_OK; }
```

GetInterfaceSafetyOptions()와 SetInterfaceSafetyOptions()함수는 IObjectSafety에서 선언된다. 이 함수들은 객체가 대본에 적합하다는것을 알리기 위하여 실현한다.

그러면 main.cpp를 고찰하자.

```
#include <qaxfactory.h>
#include "axbouncer.h"
QAXFACTORY_DEFAULT(AxBouncer,
    "{5e2461aa-a3e8-4f7a-8b04-307459a4c08c}",
    "{533af11f-4899-43de-8b7f-2ddf588d1015}",
    "{772c14a5-a840-4023-b79d-19549ece0cd9}",
    "{dbce1e56-70dd-4f74-85e0-95c65d86254d}",
    "{3f3db5e0-78ff-4e35-8a5d-3d3b96c83e09}")
int main()
{
    return 0;
}
```

QAXFACTORY_DEFAULT()마크로는 ActiveX조종요소를 반출한다. 오직 하나의 조종요소만 반출하는 ActiveX봉사기들에 이 마크로를 사용한다. 그렇지 않으면 QAxFactory의 파생클라스를 만들고 QAXFACTORY_EXPORT()라는 마크로를 사용해야 한다. 이 절의 다음 실례는 그 수행방법을 보여준다.

QAXFACTORY_DEFAULT()의 첫째 인수는 반출하려는 Qt클라스의 이름이며 또한 조종요소가 반출되는 이름이다. 그밖에 5개 인수는 클라스ID, 대면부ID, 사건대면부ID, 형서고ID 그리고 응용프로그람ID이다. guidgen이나 uuidgen과 같은 표준도구를 사용하여 이 식별자들을 생성한다.

봉사기는 서고이므로 실제의 main()함수를 요구하지 않는다. 아직은 련결프로그람을 안정시키기 위해 가실현을 제공한다.

여기에 프로쎄스내 ActiveX봉사기용의 .pro파일이 있다.

```
TEMPLATE = lib
CONFIG += activeqt dll
HEADERS = axbouncer.h \
    objectsafetyimpl.h
SOURCES = axbouncer.cpp \
    main.cpp \
```

```
    objectsafetyimpl.cpp
RC_FILE = qaxserver.rc
DEF_FILE = qaxserver.def
```

.pro파일에서 참고한 qaxserver.rc와 qaxserver.def파일은 Qt의 extensions\activeqt\control등록부로부터 복사할수 있는 표준파일들이다.

makefile 혹은 qmake가 생성한 Visual C++프로젝트파일은 Windows등록자료기지에 봉사기를 등록하기 위한 규칙들을 포함한다. 말단사용자콤퓨터들에 봉사기를 등록하기 위하여 모든 Windows체계들에서 regsvr32도구를 리용할수 있다.

그다음 <object>꼬리표를 리용하여 HTML폐지에 Bouncer부분품을 포함할수 있다.

```
<object id="AxBouncer"
  classid="clsid:5e2461aa-a3e8-4f7a-8b04-307459a4c08c">
  <b>The ActiveX control is not available. Make sure you have built and registered the
       component server.</b>
</object>
```

처리부들을 호출하는 단추들을 창조할수 있다.

```
<input type="button" value="Start" onClick="AxBouncer.start()">
<input type="button" value="Stop" onClick="AxBouncer.stop()">
```

그리고 다른 ActiveX조종요소처럼 JavaScript나 VBScript를 사용하여 창문부품을 조작할수 있다.

마지막 실례는 스크립트가능한 Address Book응용프로그람이다. 이 응용프로그람은 표준 Qt/Windows응용프로그람 혹은 프로쎄스외ActiveX봉사기로 작업할수 있다. 두번째 경우의 가능성은 가령 Visual Basic를 사용하여 응용프로그람을 스크립트하게 한다.

```
class AddressBook : public QMainWindow
{
    Q_OBJECT
    Q_PROPERTY(int count READ count)
public:
    AddressBook(QWidget *parent = 0, const char *name = 0);
    ~AddressBook();
    int count() const;
public slots:
    ABItem *createEntry(const QString &contact);
    ABItem *findEntry(const QString &contact) const;
    ABItem *entryAt(int index) const;
    ...
```

```
};
```

AddressBook창문부품은 응용프로그람의 기본창문이다. 그것이 제공하는 속성과 처리부들은 스크립팅에 사용할수 있다.

```
class ABItem : public QObject, public QListViewItem
{
    Q_OBJECT
    Q_PROPERTY(QString contact READ contact WRITE setContact)
    Q_PROPERTY(QString address READ address WRITE setAddress)
    Q_PROPERTY(QString phoneNumber READ phoneNumber WRITE setPhoneNumber)
public:
    ABItem(QListView *listView);
    void setContact(const QString &contact);
    QString contact() const { return text (0); }
    void setAddress(const QString &address);
    QString address() const { return text(1); }
    void setPhoneNumber(const QString &number);
    QString phoneNumber() const { return text(2); }
public slots:
    void remove();
};
```

ABItem클라스는 주소록의 한개 항목을 표시한다. 이것은 QListViewItem을 계승하므로 QListView에 표시할수 있으며 QObject를 계승하므로 COM객체로서 반출될수 있다.

```
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    if (!QAxFactory::isServer()) {
       AddressBook addressBook;
       app.setMainWidget(&addressBook);
       addressBook.show();
       return app.exec();
    }
    return app.exec();
}
```

main()에서는 응용프로그람이 독립적으로 실행되고있는가 봉사기로 실행되고있는가를 검사한다. activex지령행선택은 봉사기로 실행하게 한다. 응용프로그람이 봉사기로 실행되지 않으

면 기본창문부품을 창조하고 보통 독립적인 Qt응용프로그람에서 수행한것처럼 그것을 표시한다.

-activex와 함께 ActiveX봉사기들은 다음과 같은 지령행추가선택들을 인식한다.

· -regserver는 체계등록에 봉사기를 등록한다.

· -unregserver는 체계등록으로부터 봉사기등록을 해제한다.

· -dumpidl*file*은 봉사기의 IDL을 지정된 파일에 써넣는다.

응용프로그람을 봉사기로 실행하는 경우에 AddressBook와 ABItem클라스들을 COM부분품들로 반출할 필요가 있다. 즉

```
    QAXFACTORY_EXPORT(ABFactory, "{2b2b6f3e-86cf-4c49-9df5-80483b47f17b}",
        "{8e827b25-148b-4307-ba7d-23f275244818}")
```

QAXFACTORY_EXPORT()마크로는 COM객체들을 창조하기 위한 공장을 반출한다. 두가지형의 COM객체들을 반출하려고 하므로 이전의 실례에서와 같이 단순히 QAXFACTORY_DEFAULT()를 사용할수 없다.

QAXFACTORY_EXPORT()의 첫째인수는 응용프로그람의 COM객체들을 제공하는 QAxFactory클라스의 이름이다. 다른 2개 인수는 형서고ID와 응용프로그람ID이다.

```
class ABFactory : public QAxFactory
{
public:
    ABFactory(const QUuid &lib, const QUuid &app);
    QStringList featureList() const;
    QWidget *create(const QString &key, QWidget *parent, const char *name);
    QUuid classID(const QString &key) const;
    QUuid interfaceID(const QString &key) const;
    QUuid eventsID(const QString &key) const;
    QString exposeToSuperClass(const QString &key) const;
};
```

ABFactory클라스는 QAxFactory을 계승하며 AddressBook클라스를 ActiveX조종요소로, ABItem클라스를 COM부분품으로 반출하기 위한 가상함수들을 재정의한다.

```
ABFactory::ABFactory(const QUuid &lib, const QUuid &app) : QAxFactory(lib, app) {}
```

ABFactory구성자는 단순히 기초클라스 구성자에 그의 2개 파라메터를 넘긴다.

```
QStringList ABFactory::featureList() const
{
    return QStringList() << "AddressBook" << "ABItem";
}
```

featureList()함수는 공장이 제공하는 COM부분품들의 목록을 돌려준다.

```
QWidget *ABFactory::create(const QString &key, QWidget *parent, const char *name)
{
    if (key == "AddressBook")
        return new AddressBook(parent, name);
    else
        return 0;
}
```

create()함수는 ActiveX조종요소의 실례를 창조한다. 사용자들이 ABItem객체들을 창조하기를 바라지 않으므로 ABItem용의 null지적자를 돌려준다. 또한 create()돌림값형은 QWidget *이다. 이것은 ActiveX조종요소가 아닌 COM객체를 돌려주는것을 방지한다.

```
QUuid ABFactory::classID(const QString &key) const
{
    if (key == "AddressBook")
        return QUuid("{588141ef-110d-4beb-95ab-ee6a478b576d}");
    else if (key == "ABItem")
        return QUuid("{bc82730e-5f39-4e5c-96be-461c2cd0d282}");
    else return QUuid();
}
```

classId()함수는 공장에 의해 반출된 모든 클라스들의 클라스ID를 돌려준다.

```
QUuid ABFactory::interfaceID(const QString &key) const
{
    if (key == "AddressBook")
        return QUuid("{718780ec-b30c-4d88-83b3-79b3d9e78502}");
    else if (key == "ABItem")
        return QUuid("{c8bc1656-870e-48a9-9937-fbe1ceff8b2e}");
    else
        return QUuid();
}
```

interfaceID()함수는 공장에 의해 반출된 클라스들의 대면부ID를 돌려준다.

```
QUuid ABFactory::eventsID(const QString &key) const
{
    if (key == "AddressBook")
        return QUuid("{0a06546f-9f02-4f14-a269-d6d56ffeb861}");
    else if (key == "ABItem")
        return QUuid("{105c6b0a-3fc7-460b-ae59-746d9d4b1724}");
```

```
    else
        return QUuid();
}
```

eventsId()함수는 공장에 의해 반출된 클래스들의 사건대면부ID를 돌려준다.

```
QString ABFactory::exposeToSuperClass(const QString &key) const { return key; }
```

기정으로 ActiveX조종요소들은 자기뿐아니라 QWidget까지의 웃준위클라스에 이르기까지의 속성, 신호, 처리부들을 의뢰기들에 공개한다. exposeToSuperClass()함수를 재정의하여 공개하려는 제일 높은 준위의 기초클라스(계승나무에서)를 돌려준다.

여기서는 부분품의 클라스이름(AddressBook 혹은 ABItem)을 반출하려는 제일 웃준위의 기초클라스로서 돌려준다. 이것은 AddressBook와 ABItem의 웃준위클라스들에서 정의된 속성, 신호, 처리부들이 반출되지 않는다는것을 의미한다.

다음은 프로쎄스외ActiveX봉사기의 .pro파일이다.

```
CONFIG += activeqt
HEADERS = abfactory.h \
    abitem.h\
    addressbook.h\
    editdialog.h
SOURCES = abfactory.cpp\
    abitem.cpp\
    addressbook.cpp\
    editdialog.cpp\
    main.cpp
RC_FILE = qaxserver.rc
```

.pro파일에서 참고한 qaxserver.rc파일은 표준파일로서 Qt의 extensions\activeqt\control등록부에서 복사할수 있다..

Address Book봉사기를 사용하는 Visual Basic프로젝트용의 vb등록부의 실례를 보시오.

이로서 ActiveQt틀거리를 개괄하였다. Qt배포물에는 추가적인 실례들과 QAxContainer와 QAxServer모듈의 건설방법과 일반적인 호상운영문제를 해결하는 방법에 대한 정보들이 포함되여있다.

제3절. 쎄숀관리

X11에서 로그아우트할 때 일부 창문관리기들은 쎄숀을 보관하려고 하는가를 묻는다. Yes라고 대답하면 실행중에 있던 응용프로그람들은 다음번에 로그인할 때 자동적으로 재기동하고 이전에 로그아우트할 때와 같은 상태(화면위치)를 가진다.

쎄숀의 보관과 복귀를 고려하는 X11에 고유한 부분품을 쎄숀관리기(session manager)라고 부른다. Qt응용프로그람이 쎄숀관리기를 인식하게 하기 위하여 QApplication::saveState()을 재정

의하고 거기에 응용프로그람의 상태를 보관한다.

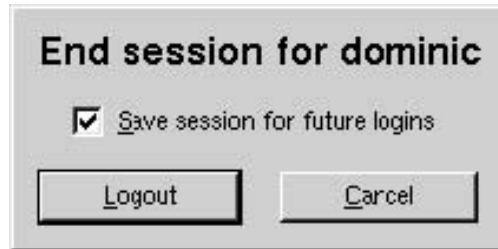



그림 18-4. KDE에서 로그아우트

Windows 2000과 XP(그리고 일부 Unix체계들)는 동면이라는 다른 기구를 제공한다. 사용자가 콤퓨터를 동면상태에 넣을 때 조작체계는 단순히 콤퓨터의 기억기를 디스크에 출력하고 기동시에 그것을 재적재한다. 응용프로그람들은 아무것도 할 필요가 없고 지어는 이런 일에 대하여 알 필요가 없다.

QApplication::commitData()을 재정의하여 사용자가 체계완료(shutdown)를 시작할 때 체계완료직전에 조종을 얻을수 있다. 이것은 보관하지 않은 자료를 보관하고 필요하다면 사용자와 교제할수 있게 한다. 이것은 X11와 Windows에서 같은 방법으로 작업한다.

쎄숀을 인식하는 TicTacToe응용프로그람의 코드를 통하여 쎄숀관리를 고찰한다. 우선 main()함수를 고찰한다.

```
int main(int argc, char *argv[])
{
    Application app(argc, argv);
    TicTacToe tic(0, "tic");
    app.setTicTacToe(&tic);
    tic.show();
    return app.exec();
}
```

Application객체를 창조한다. Application클라스는 QApplication을 계승하며 commitData()와 saveState()를 둘다 재정의하여 쎄숀관리를 유지한다.

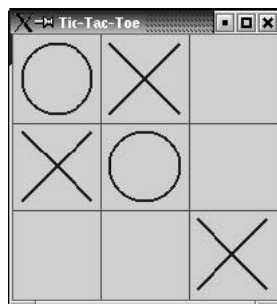



그림 18-5. TicTacToe응용프로그람

다음으로 TicTacToe창문부품을 창조하고 Application객체에 그것을 알리고 표시한다. TicTacToe창문부품을 tic라고 부른다. 쎄숀관리기가 창문들의 크기와 위치들을 되살리기를 바란다면 제일 웃준위창문부품들에 유일이름을 주어야 한다.

여기에 Application클라스의 정의가 있다.

```
class Application : public QApplication
{
    Q_OBJECT
public:
    Application(int &argc, char *argv[]);
    void setTicTacToe(TicTacToe *tic);
    void commitData(QSessionManager &sessionManager);
    void saveState(QSessionManager &sessionManager);
private:
    TicTacToe *ticTacToe;
};
```

Application클라스는 TicTacToe창문부품의 지적자를 비공개변수로서 보유한다.

```
void Application::saveState(QSessionManager &sessionManager)
{
    QString fileName = ticTacToe->saveState();
    QStringList discardCommand;
    discardCommand << "rm" << fileName;
    sessionManager.setDiscardCommand(discardCommand);
}
```

X11에서 saveState()함수는 쎄숀관리기가 응용프로그람의 상태를 보관하려고 할 때 호출된다. 이 함수는 다른 가동환경에서도 물론 사용할수 있지만 절대로 호출되지 않는다. QSessionManager파라메터는 우리가 쎄숀관리기와 교제하게 한다.

우선 파일에 TicTacToe창문부품의 상태를 보관하겠는가를 묻는다. 그다음 쎄숀관리기의 포기지령을 설정한다. 포기지령(discard command)은 현재의 상태를 고려하여 보관된 지령을 삭제하기 위하여 쎄숀관리기가 실행해야 할 지령이다. 이 실례에서는 그것을

```
rm file
```

로 설정한다.

여기서 file은 쎄숀용의 보관상태를 포함하는 파일이름이고 rm은 파일들을 삭제하는 표준 Unix지령이다.

또한 쎄숀관리기는 재기동지령을 가진다. 재기동지령(restart command)은 응용프로그람을 재기동할 때 쎄숀관리기가 실행하여야 할 지령이다. 기정으로 Qt는 다음의 재기동지령을 제

공한다.

appname -session id_key

첫부분인 *appname*은 argv[0]으로부터 끌어낸다. id부분은 쎄숀관리기가 제공하는 쎄숀ID로서 각이한 응용프로그람들과 같은 응용프로그람의 각이한 실행에서 유일하다는것을 담보한다. *key*부분은 상태가 보관된 시간을 유일하게 식별하기 위하여 추가된다. 여러가지 리유로 쎄숀관리기는 같은 쎄숀동안에 saveState()를 여러번 호출할수 있고 각이한 상태를 구별해야 한다.

현존 쎄숀관리기들의 제한으로 인하여 응용프로그람이 정확히 재기동하기를 바란다면 응용프로그람의 등록부가 PATH환경변수에 있는가 확인해야 한다. 특히 자체의 TicTacToe실례를 요구한다면 가령 그것을 /usr/bin에 설치하고 tictactoe를 실행해야 한다.

TicTacToe를 비롯한 단순한 응용프로그람들에서 재기동지령의 추가지령행인수로서 상태를 보관할수 있다. 례를 들면

```
tictactoe  -state  OX-XO-X-O
```

이것은 파일에 자료를 보관하고 파일을 삭제하기 위한 포기지령의 제공으로부터 벗어나게 한다.

```
void  Application::commitData(QSessionManager  &sessionManager)
{
    if  (ticTacToe->gameInProgress()  &&  sessionManager.allowsInteraction())  {
        int  ret  =  QMessageBox::warning(ticTacToe,  tr("Tic-Tac-Toe"),
                tr("The  game  hasn't  finished.\nDo  you  really  want  to  quit?"),
                QMessageBox::Yes  |  QMessageBox::Default,
                QMessageBox::No  |  QMessageBox::Escape);
        if  (ret  ==  QMessageBox::Yes)
                sessionManager.release();
        else sessionManager.cancel();
    }
}
```

commitData()함수는 사용자가 로그아우트할 때 호출된다. 이 함수를 재정의하여 통보창을 펼치고 사용자에게 있을수 있는 자료손실에 대하여 경고한다. 기정실현은 제일 웃준위창문부품들을 모두 닫는다. 이것은 사용자가 제목띠에서 ×단추를 찰칵함으로써 창문들을 하나씩 닫을 때와 같은 결과가 생기게 한다. 3장에서 closeEvent()를 재정의하여 이것을 포착하고 통보창을 펼치는 방법을 보았다.

이 실례에서는 commitData()를 재정의하고 통보창을 펼치여 오락을 진행하며 쎄숀관리기가 사용자와의 교제를 허용한다면 사용자에게 로그아우트를 확인할것을 요구한다. 사용자가 *Yes*를 찰칵하면 release()를 호출하여 쎄숀관리기에게 로그아우트를 계속하겠는가고 통보하고 사용자가 *No*를 찰칵하면 cancel()을 호출하여 로그아우트를 취소한다.

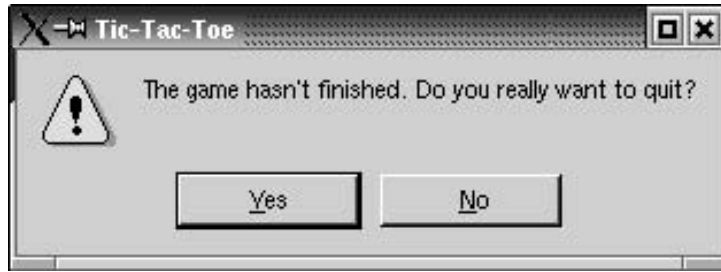



그림 18-6. "Do you really want to quit?"

이제는 TicTacToe클라스를 고찰하자.

```
class TicTacToe : public QWidget
{
        Q_OBJECT
public:
        TicTacToe(QWidget *parent = 0, const char *name = 0);
        QSize sizeHint() const;
        bool gameInProgress() const;
        QString saveState() const;
protected:
        void paintEvent(QPaintEvent *event);
        void mousePressEvent(QMouseEvent *event);
private:
        enum { Empty ='-', Cross = 'X', Nought = 'O' };
        void clearBoard();
        void restoreState();
        QString sessionFileName() const;
        QRect cellRect(int row, int col) const;
        int cellWidth() const { return width() / 3; }
        int cellHeight() const { return height() / 3; }
        char board[3][3];int turnNumber;
};
```

TicTacToe클라스는 QWidget를 계승하며 sizeHint(), paintEvent(), mousePressEvent()를 재정의한다. 또한 Application에서 사용한 gameInProgress()와 saveState()함수를 제공한다.

```
TicTacToe::TicTacToe(QWidget *parent, const char *name) : QWidget(parent, name)
{
    setCaption(tr("Tic-Tac-Toe"));
    clearBoard();
```

```
    if (qApp->isSessionRestored())
        restoreState();
}
```

구성자에서는 장기판을 지우며 응용프로그람이 –session선택으로 호출되였다면 비공개함수 restoreState()를 호출하여 낡은 쎄숀을 재적재한다.

```
void TicTacToe::clearBoard()
{
    for (int row = 0; row < 3; ++row) {
        for (int col = 0; col < 3; ++col) {
            board[row][col] = Empty;
        }
    }
    turnNumber = 0;
}
```

clearBoard()에서는 모든 세포들을 지우고 turnNumber를 0으로 설정한다.

```
QString TicTacToe::saveState() const
{
    QFile file(sessionFileName());
    if (file.open(IO_WriteOnly)) {
        QTextStream out(&file);
        for (int row = 0;row < 3;++row) {
            for (int col = 0;col < 3;++col) {
                out << board[row] [col];
            }
        }
    }
    return file.name();
}
```

saveState()에서는 장기판의 상태를 디스크에 써넣는다. 형식은 간단하며 교차일 때 'X', 령일 때 'O', 그리고 빈 세포일 때 '-'이다.

```
QString TicTacToe::sessionFileName() const
{
    return QDir::homeDirPath() + "/.tictactoe_" + qApp->sessionId() +"_"
        + qApp->sessionKey();
}
```

```
}
```

sessionFileName()비공개함수는 현재 쎄숀ID와 쎄숀열쇠용의 파일이름을 돌려준다. 이 함수는 saveState()와 restoreState()에 모두 쓰인다. 파일이름은 쎄숀ID와 쎄숀열쇠로부터 끌어낸다.

```
void TicTacToe::restoreState()
{
    QFile file(sessionFileName());
    if (file.open(IO_ReadOnly)) {
        QTextStream in(&file);
        for (int row = 0; row < 3; ++row) {
            for (int col = 0; col < 3; ++col) {
                in >> board[row][col];
                if (board[row][col] != Empty)
                    ++turnNumber;
            }
        }
    }
    repaint();
}
```

restoreState()에서는 되살아나는 쎄숀에 대응하는 파일을 적재하고 장기판을 그 정보로 채운다. 장기판에서 *X*'와 *O*'들의 수로부터 turnNumber의 값을 끌어낸다.

TicTacToe구성자에서 QApplication::isSessionRestored()이 true를 돌려주면 restoreState()를 호출한다. 그러한 경우에 sessionId()와 sessionKey()는 응용프로그람의 상태를 보관할 때와 같은 값을 돌려주므로 sessionFileName()은 그 쎄숀의 파일이름을 돌려준다.

우리는 항상 로그인하고 로그아우트할 필요가 있으므로 쎄숀관리의 시험과 오유수정은 실패할수 있다. 이것을 피하는 한가지 방법은 X11에서 제공된 표준xsm편의프로그람을 사용하는것이다. xsm을 처음으로 기동할 때 쎄숀관리기창문과 말단을 펼친다. 말단으로부터 기동하는 응용프로그람은 일반적으로 체계범위의 쎄숀관리기대신에 xsm을 쎄숀관리기로 사용한다. 그다음 xsm의 창문을 사용하여 쎄숀을 끝내고 재기동하고 혹은 포기할수 있으며 응용프로그람이 제대로 동작하는가 확인할수 있다.

참고문헌

Jusmin Blanchette, Mark Summerfield 《C++ GUI Programming with Qt 3》 Prentice Hall, 2004.

이 책은 콤퓨터를 전공으로 하는 교원, 연구사, 대학생들을 위한 참고서이다.

Qt프로그람개발법

집 필	한영철,홍이철,문홍남	**심 사**	박사 정강호, 박사 정경옥, 김영일, 문창혁
편 집	차현옥	**교 정**	서금석
장 정	서경애	**콤퓨터편성**	여은정
낸 곳	교육위원회 교육정보쎈터	**인쇄소**	
인 쇄		**발 행**	

교-09-1817　　　　부　　　　값　　　원